VISIT…

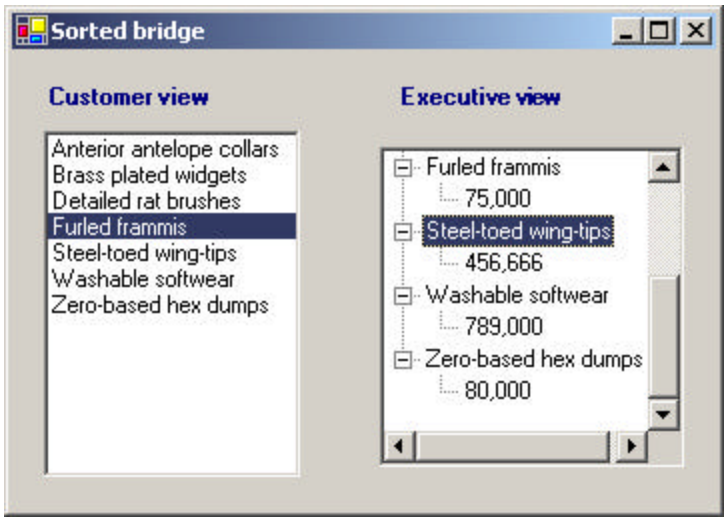


Figure 15-4– Another display using a Bridge to a tree list

Windows Forms as Bridges

The .NET visual control is itself an ideal example of a Bridge pattern implementation. A Control is a reusable software component that can be manipulated visually in a builder tool. All of the C# controls support a query interface that enables builder programs to enumerate their properties and display them for easy modification. Figure 15-5 is a screen from Visual Studio.NET displaying a panel with a text field and a check box. The builder panel to the right shows how you can modify the properties of either of those components using a simple visual interface.

Copyright © , 2002 by James W Cooper

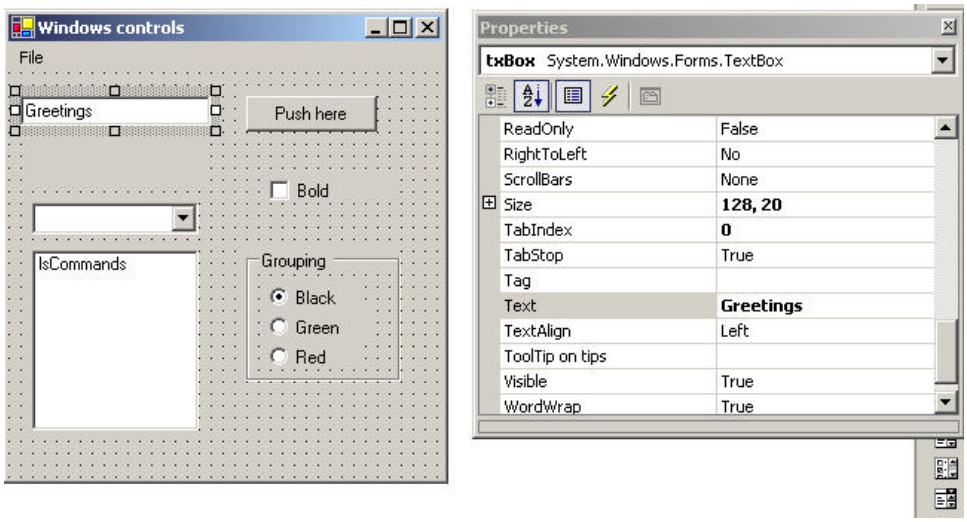


Figure 15-5 – A screen from Visual Studio.NET showing a properties interface. The property lists are effectively implemented using a Bridge pattern.

In other words, all ActiveX controls have the same interface used by the Builder program, and you can substitute any control for any other and still manipulate its properties using the same convenient interface. The actual program you construct uses these classes in a conventional way, each having its own rather different methods, but from the builder's point of view, they all appear to be the same.

Consequences of the Bridge Pattern

1. The Bridge pattern is intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. This can prevent you from recompiling a complicated set of user interface modules and only require that you recompile the bridge itself and the actual end display class.
2. You can extend the implementation class and the bridge class separately, and usually without much interaction with each other.

Copyright © , 2002 by James W Cooper

3. You can hide implementation details from the client program much more easily.

Thought Question

In plotting a stock's performance, you usually display the price and price-earnings ratio over time, whereas in plotting a mutual fund, you usually show the price and the earnings per quarter. Suggest how you can use a Bridge to do both.

Programs on the CD-ROM

`\Bridge\BasicBridge`	bridge from list to grid
`\Bridge\SortBridge`	sorted bridge

Copyright © , 2002 by James W Cooper

16. The Composite Pattern

Frequently programmers develop systems in which a component may be either an individual object or a collection of objects. The Composite pattern is designed to accommodate both cases. You can use the Composite to build part-whole hierarchies or to construct data representations of trees. In summary, a composite is a collection of objects, any one of which may be either a composite or just a primitive object. In tree nomenclature, some objects may be nodes with additional branches and some may be leaves.

The problem that develops is the dichotomy between having a single, simple interface to access all the objects in a composite and the ability to distinguish between nodes and leaves. Nodes have children and can have children added to them, whereas leaves do not at the moment have children, and in some implementations they may be prevented from having children added to them.

Some authors have suggested creating a separate interface for nodes and leaves where a leaf could have the methods, such as the following.

```
public string getName();
public float getValue();
```

And a node could have the additional methods.

```
public ArrayList elements();
public Node getChild(string nodeName);
public void add(Object obj);
public void remove(Object obj);
```

This then leaves us with the programming problem of deciding which elements will be which when we construct the composite. However, *Design Patterns* suggests that each element should have the *same* interface, whether it is a composite or a primitive element. This is easier to

Copyright © , 2002 by James W Cooper

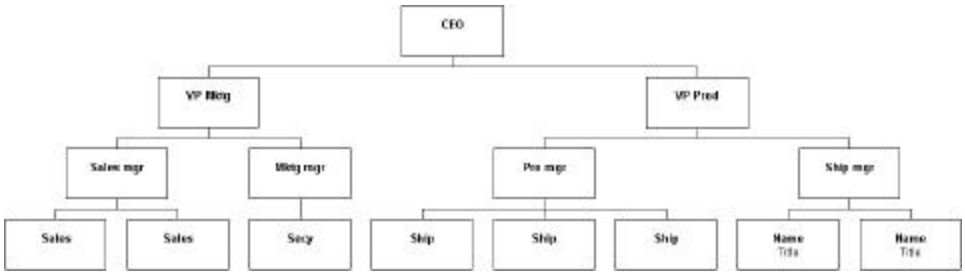
accomplish, but we are left with the question of what the *getChild* operation should accomplish when the object is actually a leaf.

C# can make this quite easy for us, since every node or leaf can return an ArrayList of the child nodes. If there are no children, the count property returns zero. Thus, if we simply obtain the ArrayList of child nodes from each element, we can quickly determine whether it has any children by checking the *count* property.

Just as difficult is the issue of adding or removing leaves from elements of the composite. A nonleaf node can have child-leaves added to it, but a leaf node cannot. However, we would like all of the components in the composite to have the same interface. We must prevent attempts to add children to a leaf node, and we can design the leaf node class to raise an error if the program attempts to add to such a node.

An Implementation of a Composite

Let's consider a small company. It may have started with a single person who got the business going. He was, of course, the CEO, although he may have been too busy to think about it at first. Then he hired a couple of people to handle the marketing and manufacturing. Soon each of them hired some additional assistants to help with advertising, shipping, and so forth, and they became the company's first two vice-presidents. As the company's success continued, the firm continued to grow until it has the organizational chart in Figure 16-1.



Copyright © , 2002 by James W Cooper

Figure 16-1 – A typical organizational chart

Computing Salaries

If the company is successful, each of these company members receives a salary, and we could at any time ask for the cost of the control span of any employee to the company. We define this control cost as the salary of that person and those of all subordinates. Here is an ideal example for a composite.

- The cost of an individual employee is simply his or her salary (and benefits).
- The cost of an employee who heads a department is his or her salary plus those of subordinates.

We would like a single interface that will produce the salary totals correctly whether the employee has subordinates or not.

```
float getSalaries();    //get salaries of all
```

At this point, we realize that the idea of all Composites having the same standard method names in their interface is probably naïve. We'd prefer that the public methods be related to the kind of class we are actually developing. So rather than have generic methods like *getValue,* we'll use *getSalaries.*

The Employee Classes

We could now imagine representing the company as a Composite made up of nodes: managers and employees. It would be possible to use a single class to represent all employees, but since each level may have different properties, it might be more useful to define at least two classes: Employees and Bosses. Employees are leaf nodes and cannot have employees under them. Bosses are nodes that may have employee nodes under them.

Copyright © , 2002 by James W Cooper

We'll start with the AbstractEmployee class and derive our concrete employee classes from it.

```
public interface AbstractEmployee       {
       float getSalary();               //get current salary
       string getName();                //get name
       bool isLeaf();                   //true if leaf
       void add(string nm, float salary);      //add subordinate
       void add(AbstractEmployee emp);         //add subordinate
       IEnumerator getSubordinates();          //get subordinates
       AbstractEmployee getChild();            //get child
       float getSalaries();             //get sum of salaries
}
```

In C# we have a built-in enumeration interface called IEnumerator. This interface consists of these methods.

```
bool MoveNext();        //False if no more left
object Current()        //get current object
void Reset();           /move to first
```

So we can create an AbstractEmployee interface that returns an Enumerator. You move through an enumeration, allowing for the fact that it might be empty, using the following approach.

```
e.Reset();
while (e.MoveNext()) {
  Emp = (Employee)e.Current();
  //..do computation..
}
```

This Enumerator may, of course, be empty and can thus be used for both nodes and leaves of the composite.

Our concrete Employee class will store the name and salary of each employee and allow us to fetch them as needed.

```
public class Employee :AbstractEmployee       {
       protected float salary;
       protected string name;
       protected ArrayList subordinates;
       //------
```

Copyright © , 2002 by James W Cooper

```
public Employee(string nm, float salry)                {
       subordinates = new ArrayList();
       name = nm;
       salary = salry;
}
//------
public float getSalary() {
       return salary;
}
//------
public string getName() {
       return name;
}
//------
public bool isLeaf() {
       return subordinates.Count == 0;
}
//------
public virtual AbstractEmployee getChild() {
       return null;
}
```

The Employee class must have concrete implementations of the *add, remove, getChild,* and *subordinates* classes. Since an Employee is a leaf, all of these will return some sort of error indication. The subordinates method could return a null, but programming will be more consistent if *subordinates* returns an empty enumeration.

```
public IEnumerator getSubordinates() {
       return subordinates.GetEnumerator ();
}
```

The *add* and *remove* methods must generate errors, since members of the basic Employee class cannot have subordinates. We throw an Exception if you call these methods in the basic Employee class.

```
public virtual void add(string nm, float salary) {
  throw new Exception(
       "No subordinates in base employee class");
}
//------
public virtual void add(AbstractEmployee emp) {
```

Copyright © , 2002 by James W Cooper

```
            throw new Exception(
            "No subordinates in base employee class");
      }
```

The Boss Class

Our Boss class is a subclass of Employee and allows us to store subordinate employees as well. We'll store them in an ArrayList called *subordinates* and return them through an enumeration. Thus, if a particular Boss has temporarily run out of Employees, the enumeration will just be empty.

```
public class Boss:Employee {
      public Boss(string name, float salary):base(name,salary) {}
      //------
      public override void add(string nm, float salary) {
            AbstractEmployee emp = new Employee(nm,salary);
            subordinates.Add (emp);
      }
      //------
      public override void add(AbstractEmployee emp){
            subordinates.Add(emp);
      }
      //------
```

If you want to get a list of employees of a given supervisor, you can obtain an Enumeration of them directly from the ArrayList. Similarly, you can use this same ArrayList to returns a sum of salaries for any employee and his subordinates.

```
public float getSalaries() {
      float sum;
      AbstractEmployee esub;
      //get the salaries of the boss and subordinates
      sum = getSalary();
      IEnumerator enumSub = subordinates.GetEnumerator() ;
      while (enumSub.MoveNext())  {
            esub = (AbstractEmployee)enumSub.Current;
            sum += esub.getSalaries();
      }
  return sum;
```

Copyright © , 2002 by James W Cooper

```
}
```

Note that this method starts with the salary of the current Employee and then calls the *getSalaries()* method on each subordinate. This is, of course, recursive, and any employees who have subordinates will be included. A diagram of these classes is shown in Figure 16-2.

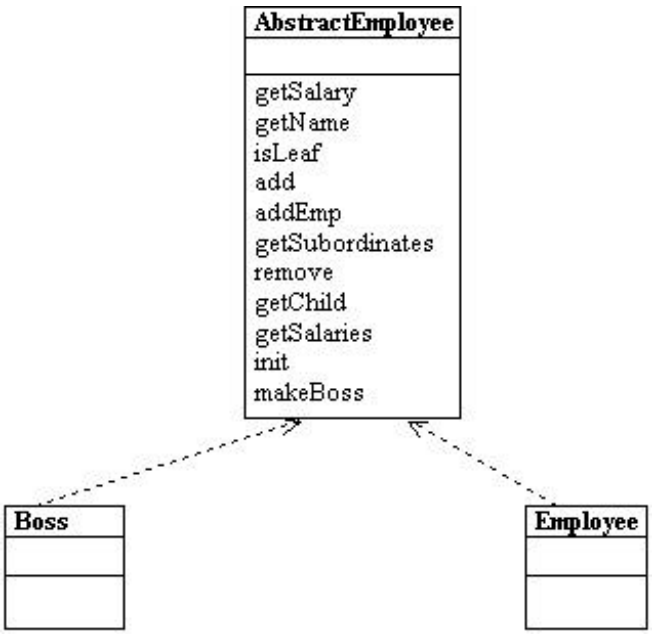


Figure 16-2 – The AbstractEmployee class and how Employee and Boss are derived from it

Building the Employee Tree

We start by creating a CEO Employee and then add his subordinates and their subordinates, as follows.

```
private void buildEmployeeList() {
```

Copyright © , 2002 by James W Cooper

```
        prez = new Boss("CEO", 200000);
        marketVP = new Boss("Marketing VP", 100000);
        prez.add(marketVP);
        salesMgr = new Boss("Sales Mgr", 50000);
        advMgr = new Boss("Advt Mgr", 50000);
        marketVP.add(salesMgr);
        marketVP.add(advMgr);
        prodVP = new Boss("Production VP", 100000);
        prez.add(prodVP);
        advMgr.add("Secy", 20000);
        //add salesmen reporting to sales manager
        for (int i = 1; i<=5; i++){
               salesMgr.add("Sales" + i.ToString(),
                      rand_sal(30000));
        }

        prodMgr = new Boss("Prod Mgr", 40000);
        shipMgr = new Boss("Ship Mgr", 35000);
        prodVP.add(prodMgr);
        prodVP.add(shipMgr);

        for (int i = 1; i<=3; i++){
               shipMgr.add("Ship" + i.ToString(), rand_sal(25000));
        }
        for (int i = 1; i<=4; i++){
               prodMgr.add("Manuf" + i.ToString(), rand_sal(20000));
        }
}
```

Once we have constructed this Composite structure, we can load a visual TreeView list by starting at the top node and calling the *addNode()* method recursively until all the leaves in each node are accessed.

```
private void buildTree() {
        EmpNode nod;
        nod = new EmpNode(prez);
        rootNode = nod;
        EmpTree.Nodes.Add(nod);
        addNodes(nod, prez);
}
```

To simplify the manipulation of the TreeNode objects, we derive an EmpNode class which takes an instance of Employee as an argument:

Copyright © , 2002 by James W Cooper

```
public class EmpNode:TreeNode     {
       private AbstractEmployee emp;
       public EmpNode(AbstractEmployee aemp ):
                     base(aemp.getName ()) {
              emp = aemp;
       }
       //-----
       public AbstractEmployee getEmployee() {
              return emp;
       }
}
```

The final program display is shown in Figure 16-3.

Figure 16-3 – The corporate organization shown in a TreeView control

In this implementation, the cost (sum of salaries) is shown in the bottom bar for any employee you click on. This simple computation calls the *getChild()* method recursively to obtain all the subordinates of that employee.

```
private void EmpTree_AfterSelect(object sender,
```

Copyright © , 2002 by James W Cooper

```
            TreeViewEventArgs e) {
      EmpNode node;
      node = (EmpNode)EmpTree.SelectedNode;
      getNodeSum(node);
}
//------
private void getNodeSum(EmpNode node) {
      AbstractEmployee emp;
      float sum;

       emp = node.getEmployee();
       sum = emp.getSalaries();
       lbSalary.Text = sum.ToString ();
}
```

Self-Promotion

We can imagine cases where a simple Employee would stay in his current job but have new subordinates. For example, a Salesman might be asked to supervise sales trainees. For such a case, it is convenient to provide a method in the Boss class that creates a Boss from an Employee. We just provide an additional constructor that converts an employee into a boss:

```
public Boss(AbstractEmployee emp):
            base(emp.getName() , emp.getSalary())   {
}
```

Doubly Linked Lists

In the preceding implementation, we keep a reference to each subordinate in the Collection in each Boss class. This means that you can move down the chain from the president to any employee, but there is no way to move back up to find out who an employee's supervisor is. This is easily remedied by providing a constructor for each AbstractEmployee subclass that includes a reference to the parent node.

```
public class Employee :AbstractEmployee        {
      protected float salary;
      protected string name;
      protected AbstractEmployee parent;
      protected ArrayList subordinates;
```

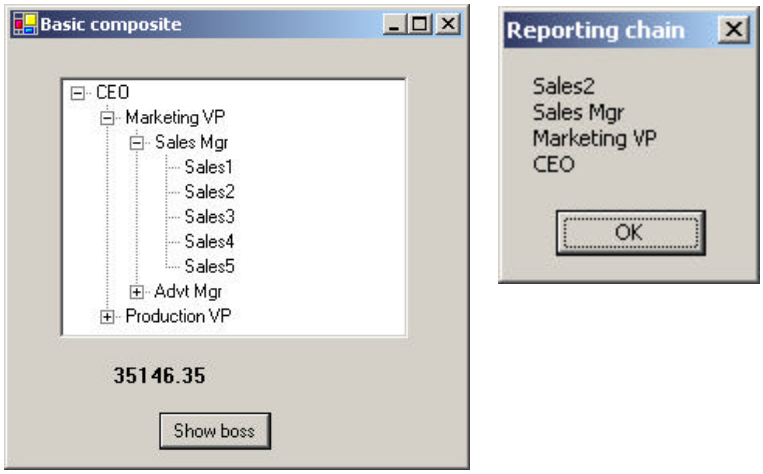
Copyright © , 2002 by James W Cooper

```
        //------
        public Employee(AbstractEmployee parnt,
                        string nm, float salry)          {
               subordinates = new ArrayList();
               name = nm;
               salary = salry;
               parent = parnt;
        }
```

Then you can quickly walk up the tree to produce a reporting chain.

```
private void btShowBoss_Click(object sender, System.EventArgs e) {
        EmpNode node;
        node = (EmpNode)EmpTree.SelectedNode;
        AbstractEmployee emp = node.getEmployee ();
        string bosses = "";
        while(emp != null) {
              bosses += emp.getName () +"\n";
                emp = emp.getBoss();
        }
        MessageBox.Show (null, bosses,"Reporting chain");
}
```

See Figure 16-4.



Copyright © , 2002 by James W Cooper

Figure 16-4– The tree list display of the composite with a display of the parent nodes on the right

Consequences of the Composite Pattern

The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.

The Composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface. On the other hand, this has the disadvantage of making your system overly general. You might find it harder to restrict certain classes where this would normally be desirable.

A Simple Composite

The intent of the Composite pattern is to allow you to construct a tree of various related classes, even though some have different properties than others and some are leaves that do not have children. However, for very simple cases, you can sometimes use just a single class that exhibits both parent and leaf behavior. In the SimpleComposite example, we create an Employee class that always contains the ArrayList *subordinates*. This collection of employees will either be empty or populated, and this determines the nature of the values that you return from the *getChild* and *remove* methods. In this simple case, we do not raise errors and always allow leaf nodes to be promoted to have child nodes. In other words, we always allow execution of the *add* method.

While you may not regard this automatic promotion as a disadvantage, in systems where there are a very large number of leaves, it is wasteful to keep a Collection initialized and unused in each leaf node. In cases where there are relatively few leaf nodes, this is not a serious problem.

Copyright © , 2002 by James W Cooper

Composites in .NET

In .NET, you will note that the *Node* object class we use to populate the TreeView is in fact just such a simple composite pattern. You will also find that the Composite describes the hierarchy of Form, Frame, and Controls in any user interface program. Similarly, toolbars are containers, and each may contain any number of other containers.

Any container may then contain components such as Buttons, Checkboxes, and TextBoxes, each of which is a leaf node that cannot have further children. They may also contain ListBoxes and grids that may be treated as leaf nodes or that may contain further graphical components. You can walk down the Composite tree using the *Controls* collection.

Other Implementation Issues

Ordering components. In some programs, the order of the components may be important. If that order is somehow different from the order in which they were added to the parent, then the parent must do additional work to return them in the correct order. For example, you might sort the original collection alphabetically and return a new sorted collection.

Caching results. If you frequently ask for data that must be computed from a series of child components, as we did here with salaries, it may be advantageous to cache these computed results in the parent. However, unless the computation is relatively intensive and you are quite certain that the underlying data have not changed, this may not be worth the effort.

Thought Questions

1. A baseball team can be considered an aggregate of its individual players. How could you use a composite to represent individual and team performance?
2. The produce department of a supermarket needs to track its sales performance by food item. Suggest how a composite might be helpful.

Copyright © , 2002 by James W Cooper

Programs on the CD-ROM

\Composite\Composite	composite shows tree
\Composite\DlinkComposite	composite that uses both child links and parent links
\Composite\SimpleComposite	Simple composite of same employee tree that allows any employee to move from leaf to node.

Copyright © , 2002 by James W Cooper

17. The Decorator Pattern

The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class. Suppose we have a program that uses eight objects, but three of them need an additional feature. You could create a derived class for each of these objects, and in many cases this would be a perfectly acceptable solution. However, if each of these three objects requires *different* features, this would mean creating three derived classes. Further, if one of the classes has features of *both* of the other classes, you begin to create complexity that is both confusing and unnecessary.

For example, suppose we wanted to draw a special border around some of the buttons in a toolbar. If we created a new derived button class, this means that all of the buttons in this new class would always have this same new border when this might not be our intent.

Instead, we create a Decorator class that *decorates* the buttons. Then we derive any number of specific Decorators from the main Decorator class, each of which performs a specific kind of decoration. In order to decorate a button, the Decorator has to be an object derived from the visual environment so it can receive paint method calls and forward calls to other useful graphic methods to the object that it is decorating. This is another case where object containment is favored over object inheritance. The decorator is a graphical object, but it contains the object it is decorating. It may intercept some graphical method calls, perform some additional computation, and pass them on to the underlying object it is decorating.

Decorating a CoolButton

Recent Windows applications such as Internet Explorer and Netscape Navigator have a row of flat, unbordered buttons that highlight themselves with outline borders when you move your mouse over them. Some Windows programmers call this toolbar a CoolBar and the buttons CoolButtons. There is no analogous button behavior in C# controls, but

Copyright © , 2002 by James W Cooper

we can obtain that behavior by *decorating* a Panel and using it to contain a button. In this case, we decorate it by drawing black and white border lines to highlight the button, or gray lines to remove the button borders.

Let's consider how to create this Decorator. *Design Patterns* suggests that Decorators should be derived from some general visual component class and then every message for the actual button should be forwarded from the decorator. This is not all that practical in C#, but if we use containers as decorators, all of the events are forwarded to the control being contained.

Design Patterns further suggests that classes such as Decorator should be abstract classes and that you should derive all of your actual working (or concrete) decorators from the Abstract class. In our implementation, we define a Decorator interface that receives the mouse and paint events we need to intercept.

```
public interface Decorator       {
       void mouseMove(object sender, MouseEventArgs e);
       void mouseEnter(object sender, EventArgs e);
       void mouseLeave(object sender, EventArgs e);
       void paint(object sender, PaintEventArgs e);
}
```

For our actual implementation, we can derive a CoolDecorator from a Panel class, and have it become the container which holds the button we are going to decorate.

Now, let's look at how we could implement a CoolButton. All we really need to do is to draw the white and black lines around the button area when it is highlighted and draw gray lines when it is not. When a MouseMove is detected over the button, the next paint event should draw the highlighted lines, and when the mouse leaves the button area, the next paint event should draw outlines in gray. We do this by setting a mouse_over flag and then forcing a repaint by calling the Refresh method.

```
public void mouseMove(object sender, MouseEventArgs e){
             mouse_over = true;
}
public void mouseEnter(object sender, EventArgs e){
```

Copyright © , 2002 by James W Cooper

```
        mouse_over = true;
        this.Refresh ();
}
public void mouseLeave(object sender, EventArgs e){
        mouse_over = false;
        this.Refresh ();
}
```

The actual paint event is the following:

```
public virtual void paint(object sender, PaintEventArgs e){
        //draw over button to change its outline
        Graphics g = e.Graphics;
        const int  d = 1;
        //draw over everything in gray first
        g.DrawRectangle(gPen, 0, 0, x2 - 1, y2 - 1);
        //draw black and white boundaries
        //if the mouse is over
        if( mouse_over) {
                g.DrawLine(bPen, 0, 0, x2 - d, 0);
                g.DrawLine(bPen, 0, 0, 0, y2 - 1);
                g.DrawLine(wPen, 0, y2 - d, x2 - d, y2 - d);
                g.DrawLine(wPen, x2 - d, 0, x2 - d, y2 - d);
        }
}
```

Handling events in a Decorator

When we construct an actual decorator containing the mouse and paint methods we show above, we have to connect the event handling system to these methods. We do this in the constructor for the decorator by creating an EventHandler class for the mouse enter and hover events and a MouseEventHandler for the move and leave events. It is important to note that the events we are catching are events on the contained button, rather than on the surrounding Panel. So, the control we add the handlers to is the button itself.

```
public CoolDecorator(Control c) {
contl = c;          //copy in control

//mouse over, enter handler
EventHandler evh = new EventHandler(mouseEnter);
```

Copyright © , 2002 by James W Cooper

```
      c.MouseHover += evh;
      c.MouseEnter+= evh;
//mouse move handler
c.MouseMove += new MouseEventHandler(mouseMove);
c.MouseLeave += new EventHandler(mouseLeave);
```

Similarly, we create a PaintEventHandler for the paint event.

```
//paint handler catches button's paint
c.Paint += new PaintEventHandler( paint);
```

Layout Considerations

If you create a Windows form containing buttons, the GUI designer automatically generates code to add that Control to the Controls array for that Window. We want to change this by adding the button to the Controls array for the new panel, adding the panel to the Controls array for the Window, and removing the button from that array. Here is the code to add the panel and remove the button in the Form initialization method:

```
//add outside decorator to the layout
//and remove the button from the layout
this.Controls.AddRange(new System.Windows.Forms.Control[] {cdec});
this.Controls.Remove (btButtonA);
```

and this is the code to add the button to the Decorator panel:

```
public CoolDecorator(Control c) {
      contl = c;          //copy in control
      //add button to controls contained in panel
      this.Controls.AddRange(new Control[] {contl});
```

Control Size and Position

When we decorate the button by putting it in a Panel, we need to change the coordinates and sizes so that the Panel has the size and coordinates of the button and the button has a location of (0, 0) within the panel. This also happens in the CoolDecorator constructor:

```
this.Location = p;
```

Copyright © , 2002 by James W Cooper

```
contl.Location =new Point(0,0);

this.Name = "deco"+contl.Name ;
this.Size = contl.Size;
x1 = c.Location.X - 1;
y1 = c.Location.Y - 1;
x2 = c.Size.Width;
y2 = c.Size.Height;
```

We also create instances of the Pens we will use in the Paint method in this constructor:

```
//create the overwrite pens
gPen = new Pen(c.BackColor, 2);   //gray pen overwrites borders
bPen = new Pen(Color.Black , 1);
wPen = new Pen(Color.White, 1);
```

Using a Decorator

This program is shown in Figure 17-1, with the mouse hovering over one of the buttons.

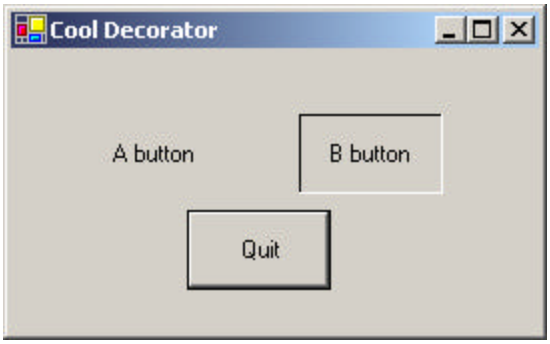


Figure 17-1 – The A button and B button are CoolButtons, which are outlined when a mouse hovers over them. Here the B button is outlined.

Multiple Decorators

Now that we see how a single decorator works, what about multiple decorators? It could be that we'd like to decorate our CoolButtons with another decoration— say, a diagonal red line.

Copyright © , 2002 by James W Cooper

This is only slightly more complicated, because we just need to enclose the CoolDecorator inside yet another decorator panel for more decoration to occur. The only real change is that we not only need the instance of the panel we are wrapping in another, but also the central object (here a button) being decorated, since we have to attached our paint routines to that central object's paint method.

So we need to create a constructor for our decorator that has both the enclosing panel and the button as Controls.

```
public class CoolDecorator :Panel, Decorator   {
       protected Control contl;
       protected Pen bPen, wPen, gPen;
       private bool mouse_over;
       protected float x1, y1, x2, y2;
//---------------------------------
       public CoolDecorator(Control c, Control baseC) {
       //the first control is the one layed out
       //the base control is the one whose paint method we extend
       //this allows for nesting of decorators
              contl = c;
              this.Controls.AddRange(new Control[] {contl});
```

Then, when we add the event handlers, the paint event handler must be attached to the base control:

```
//paint handler catches button's paint
       baseC.Paint += new PaintEventHandler( paint);
```

We make the paint method virtual so we can override it as we see below.

```
public virtual void paint(object sender, PaintEventArgs e){
       //draw over button to change its outline
       Graphics g = e.Graphics;
```

It turns out that the easiest way to write our SlashDecorator, which draws that diagonal red line, is to derive it from CoolDecorato directly. We can reuse all the base methods and extend only the paint method from the CoolDecorator and save a lot of effort.

```
public class SlashDeco:CoolDecorator    {
       private Pen rPen;
```

Copyright © , 2002 by James W Cooper

```
        //-----------------
        public SlashDeco(Control c, Control bc):base(c, bc) {
          rPen = new Pen(Color.Red , 2);
        }
        //-----------------
        public override void paint(object sender,
                     PaintEventArgs e){

              Graphics g = e.Graphics ;
              x1=0; y1=0;
              x2=this.Size.Width ;
              y2=this.Size.Height ;
              g.DrawLine (rPen, x1, y1, x2, y2);
         }
   }
```

This gives us a final program that displays the two buttons, as shown in Figure Figure 17-2. The class diagram is shown in Figure 17-3

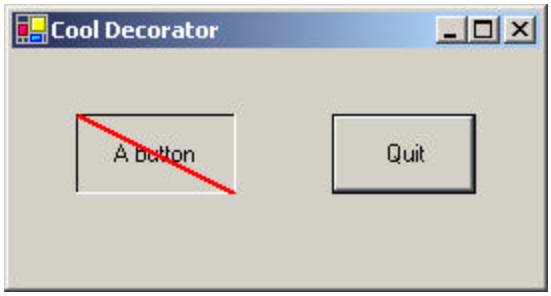


Figure 17-2 – The A CoolButton is also decorated with a SlashDecorator.

Copyright © , 2002 by James W Cooper

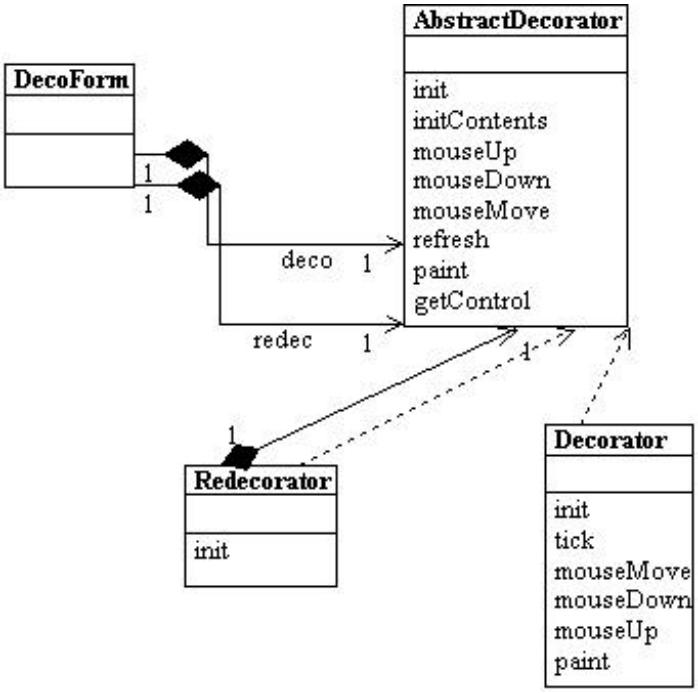


Figure 17-3 – The UML class diagram for Decorators and two specific Decorator implementations

Nonvisual Decorators

Decorators, of course, are not limited to objects that enhance visual classes. You can add or modify the methods of any object in a similar fashion. In fact, nonvisual objects can be easier to decorate because there may be fewer methods to intercept and forward. Whenever you put an instance of a class inside another class and have the outer class operate on it, you are essentially "decorating" that inner class. This is one of the most common tools for programming available in Visual Studio.NET.

Copyright © , 2002 by James W Cooper

Decorators, Adapters, and Composites

As noted in *Design Patterns*, there is an essential similarity among these classes that you may have recognized. Adapters also seem to "decorate" an existing class. However, their function is to change the interface of one or more classes to one that is more convenient for a particular program. Decorators add methods to particular instances of classes rather than to all of them. You could also imagine that a composite consisting of a single item is essentially a decorator. Once again, however, the intent is different.

Consequences of the Decorator Pattern

The Decorator pattern provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class. It also allows you to customize a class without creating subclasses high in the inheritance hierarchy. *Design Patterns* points out two disadvantages of the Decorator pattern. One is that a Decorator and its enclosed component are not identical. Thus, tests for object types will fail. The second is that Decorators can lead to a system with "lots of little objects" that all look alike to the programmer trying to maintain the code. This can be a maintenance headache.

Decorator and Façade evoke similar images in building architecture, but in design pattern terminology, the Façade is a way of hiding a complex system inside a simpler interface, whereas Decorator adds function by wrapping a class. We'll take up the Façade next.

Thought Questions

1. When someone enters an incorrect value in a cell of a grid, you might want to change the color of the row to indicate the problem. Suggest how you could use a Decorator.
2. A mutual fund is a collection of stocks. Each one consists of an array or Collection of prices over time. Can you see how a Decorator can be used to produce a report of stock performance for each stock and for the whole fund?

Copyright © , 2002 by James W Cooper

Programs on the CD-ROM

\Decorator\Cooldecorator	C#cool button decorator
\Decorator\Redecorator	C# cool button and slash decorator

Copyright © , 2002 by James W Cooper

18. The Façade Pattern

The Façade pattern is used to wrap a set of complex classes into a simpler enclosing interface. As your programs evolve and develop, they grow in complexity. In fact, for all the excitement about using design patterns, these patterns sometimes generate so many classes that it is difficult to understand the program's flow. Furthermore, there may be a number of complicated subsystems, each of which has its own complex interface.

The Façade pattern allows you to simplify this complexity by providing a simplified interface to these subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but it usually provides all the function needed for all but the most sophisticated users. These users can still, of course, access the underlying classes and methods.

Fortunately, we don't have to write a complex system to provide an example of where a Façade can be useful. C# provides a set of classes that connect to databases, using an interface called ADO.Net. You can connect to any database for which the manufacturer has provided a ODBC connection class—almost every database on the market. Let's take a minute and review how databases are used and a little about how they work.

What Is a Database?

A *database* is a series of tables of information in some sort of file structure that allows you to access these tables, select columns from them, sort them, and select rows based on various criteria. Databases usually have *indexes* associated with many of the columns in these tables, so we can access them as rapidly as possible.

Databases are used more than any other kind of structure in computing. You'll find databases as central elements of employee records and payroll systems, in travel scheduling systems, and all through product manufacturing and marketing.

Copyright © , 2002 by James W Cooper

In the case of employee records, you could imagine a table of employee names and addresses and of salaries, tax withholding, and benefits. Let's consider how these might be organized. You can imagine one table of employee names, addresses, and phone numbers. Other information that you might want to store would include salary, salary range, last raise, next raise, employee performance ranking, and so forth.

Should this all be in one table? Almost certainly not. Salary ranges for various employee types are probably invariant between employees, and thus you would store only the employee type in the employee table and the salary ranges in another table that is pointed to by the type number. Consider the data in Table 18-1.

Key	Lastname	SalaryType
1	Adams	2
2	Johnson	1
3	Smyth	3
4	Tully	1
5	Wolff	2

SalaryType	Min	Max
1	30000	45000
2	45000	60000
3	60000	75000

Table 18-1 – Employee Names and Salary Type Tables

The data in the `SalaryType` column refers to the second table. We could imagine many such tables for things like state of residence and tax values for each state, health plan withholding, and so forth. Each table will have a primary key column like the ones at the left of each table and several more columns of data. Building tables in a database has evolved to both an art and a science. The structure of these tables is referred to by their *normal form*. Tables are said to be in first, second, or third normal form, abbreviated as 1NF, 2NF, or 3NF.

- *First.* Each cell in a table should have only one value (never an array of values). (1NF)

Copyright © , 2002 by James W Cooper

- *Second.* 1NF and every non-key column is fully dependent on the key column. This means there is a one-to-one relationship between the primary key and the remaining cells in that row. (2NF)
- *Third.* 2NF and all non-key columns are mutually independent. This means that there are no data columns containing values that can be calculated from other columns' data. (3NF)

Today, nearly all databases are constructed so that all tables are in third normal form (3NF). This means that there are usually a fairly large number of tables, each with relatively few columns of information.

Getting Data Out of Databases

Suppose we wanted to produce a table of employees and their salary ranges for some planning exercise. This table doesn't exist directly in the database, but it can be constructed by issuing a query to the database. We'd like to have a table that looked like the data in Table 18-2.

Name	Min	Max
Adams	$45,000.00	$60,000.00
Johnson	$30,000.00	$45,000.00
Smyth	$60,000.00	$75,000.00
Tully	$30,000.00	$45,000.00
Wolff	$45,000.00	$60,000.00

Table 18-2 - Employee Salaries Sorted by Name

Maybe we want data sorted by increasing salary, as shown in Table 18-3.

Name	Min	Max
Tully	$30,000.00	$45,000.00
Johnson	$30,000.00	$45,000.00
Wolff	$45,000.00	$60,000.00
Adams	$45,000.00	$60,000.00
Smyth	$60,000.00	$75,000.00

Table 18-3– Employee Salaries Sorted by Magnitude

Copyright © , 2002 by James W Cooper

We find that the query we issue to obtain these tables has this form.

```
SELECT DISTINCTROW Employees.Name, SalaryRanges.Min,
SalaryRanges.Max FROM Employees INNER JOIN SalaryRanges ON
Employees.SalaryKey = SalaryRanges.SalaryKey
ORDER BY SalaryRanges.Min;
```

This language is called Structured Query Language or SQL (often pronounced "sequel"), and it is the language of virtually all databases currently available. There have been several standards issued for SQL over the years, and most PC databases support much of these ANSI standards. The SQL-92 standard is considered the floor standard, and there have been several updates since. However, none of these databases support the later SQL versions perfectly, and most offer various kinds of SQL extensions to exploit various features unique to their database.

Kinds of Databases

Since the PC became a major office tool, there have been a number of popular databases developed that are intended to run by themselves on PCs. These include elementary databases like Microsoft Works and more sophisticated ones like Approach, dBase, Borland Paradox, Microsoft Access, and FoxBase.

Another category of PC databases includes that databases intended to be accessed from a server by a number of PC clients. These include IBM DB/2, Microsoft SQL Server, Oracle, and Sybase. All of these database products support various relatively similar dialects of SQL, and thus all of them would appear at first to be relatively interchangeable. The reason they are *not* interchangeable, of course, is that each was designed with different performance characteristics involved and each with a different user interface and programming interface. While you might think that since they all support SQL, programming them would be similar, quite the opposite is true. Each database has its own way of receiving the SQL queries and its own way of returning the results. This is where the next proposed level of standardization came about: ODBC.

Copyright © , 2002 by James W Cooper

ODBC

It would be nice if we could somehow write code that was independent of the particular vendor's database that would allow us to get the same results from any of these databases without changing our calling program. If we could only write some wrappers for all of these databases so that they all appeared to have similar programming interfaces, this would be quite easy to accomplish.

Microsoft first attempted this feat in 1992 when they released a specification called Object Database Connectivity. It was supposed to be the answer for connection to all databases under Windows. Like all first software versions, this suffered some growing pains, and another version was released in 1994 that was somewhat faster as well as more stable. It also was the first 32-bit version. In addition, ODBC began to move to platforms other than Windows and has by now become quite pervasive in the PC and Workstation world. Nearly every major database vendor provides ODBC drivers.

Database Structure

At the lowest level, then, a database consists of a series of tables, each having several named columns, and some relationships between these tables. This can get pretty complicated to keep track of, and we would like to see some simplification of this in the code we use to manipulate databases.

C# and all of VisualStudio.Net use a new database access model, called ADO.NET, for ActiveX Data Objects. The design philosophy of ADO.NET is one in which you define a connection between your program and a database and use that connection sporadically, with much of the computation actually taking place in disconnected objects on your local machine. Further, ADO.NET uses XML for definition of the objects that are transmitted between the database and the program, primarily under the covers, although it is possible to access this data description using some of the built-in ADO.NET classes.

Copyright © , 2002 by James W Cooper

Using ADO.NET

ADO.NET as implemented in C# consists of a fairly large variety of interrelated objects. Since the operations we want to perform are still the same relatively simple ones, the Façade pattern will be an ideal way to manage them.

- **OleDbConnection**—This object represents the actual connection to the database. You can keep an instance of this class available but open and close the connection as needed. You must specifically close it when you are done, before it is garbage collected.
- **OleDbCommand**—This class represents a SQL command you send to the database, which may or may not return results.
- **OleDbDataAdapter**—Provides a bridge for moving data between a database and a local DataSet. You can specify an OleDbCommand, a Dataset, and a connection.
- **DataSet—**A representation of one or more database tables or results from a query on your local machine.
- **DataTable**—A single data table from a database or query
- **DataRow**—A single row in a DataTable.

Connecting to a Database

To connect to a database, you specify a connection string in the constructor for the database you want to use. For example, for an Access database, your connection string would be the following.

```
string connectionString =
      "Provider=Microsoft.Jet.OLEDB.4.0;" +
      "Data Source=" + dbName;
```

and the following makes the actual connection.

```
OleDbConnection conn =
     new OleDbConnection(connectionString);
```

Copyright © , 2002 by James W Cooper

You actually open that connection by calling the open method. To make sure that you don't re-open an already open connection, you can check its state first.

```
private void openConnection() {
       if (conn.State == ConnectionState.Closed){
              conn.Open ();
       }
}
```

Reading Data from a Database Table

To read data in from a database table, you create an ADOCommand with the appropriate Select statement and connection.

```
public DataTable openTable (string tableName) {
       OleDbDataAdapter adapter = new OleDbDataAdapter ();
       DataTable dtable = null;
       string query = "Select * from " + tableName;
       adapter.SelectCommand = new OleDbCommand (query, conn);
```

Then, you create a dataset object into which to put the results.

```
       DataSet dset = new DataSet ("mydata");
```

Then, you simply tell the command object to use the connection to fill the dataset. You must specify the name of the table to fill in the FillDataSet method, as we show here.

```
try {
       openConnection();
       adapter.Fill (dset);
}
catch(Exception e) {
       Console.WriteLine (e.Message );
}
```

The dataset then contains at least one table, and you can obtain it by index or by name and examine its contents.

Copyright © , 2002 by James W Cooper

```
        //get the table from the dataset

             dtable = dset.Tables [0];
```

Executing a Query

Executing a Select query is exactly identical to the preceding code, except the query can be an SQL Select statement of any complexity. Here we show the steps wrapped in a Try block in case there are SQL or other database errors.

```
public DataTable openQuery(string query) {
      OleDbDataAdapter dsCmd = new OleDbDataAdapter ();
      DataSet dset = new DataSet ();
//create a dataset
      DataTable dtable = null;      //declare a data table
      try {
            //create the command
            dsCmd.SelectCommand =
                  new OleDbCommand(query, conn);
      //open the connection
            openConnection();
            //fill the dataset
            dsCmd.Fill(dset, "mine");
            //get the table
            dtable = dset.Tables[0];
            //always close it
            closeConnection();
            //and return it
            return dtable;
            }
            catch (Exception e) {
                  Console.WriteLine (e.Message);
                  return null;
            }
}
```

Deleting the Contents of a Table

You can delete the contents of a table using the “Delete * from Table” SQL statement. However, since this is not a Select command, and there is

Copyright © , 2002 by James W Cooper

no local table to bridge to, you can simply use the ExecuteNonQuery method of the OleDbCommand object.

```
public void delete() {
       //deletes entire table
       conn = db.getConnection();
       openConn();
       if (conn.State == ConnectionState.Open ) {
              OleDbCommand adcmd =
              new OleDbCommand("Delete * from " + tableName, conn);
              try{
                     adcmd.ExecuteNonQuery();
                     closeConn();
              }
              catch (Exception e) {
                     Console.WriteLine (e.Message);
              }
       }
```

Adding Rows to Database Tables Using ADO.NET

The process of adding data to a table is closely related. You generally start by getting the current version of the table from the database. If it is very large, you can get only the empty table by getting just its schema. We follow these steps.

1. Create a DataTable with the name of the table in the database.
2. Add it to a dataset.
3. Fill the dataset from the database.
4. Get a new row object from the DataTable.
5. Fill in its columns.
6. Add the row to the table.
7. When you have added all the rows, update the database from the modified DataTable object.

The process looks like this.

```
DataSet dset = new DataSet(tableName);   //create the data set
```

Copyright © , 2002 by James W Cooper

```
dtable = new DataTable(tableName);   //and a datatable
dset.Tables.Add(dtable);             //add to collection
conn = db.getConnection();
openConn();                       //open the connection
OleDbDataAdapter adcmd = new OleDbDataAdapter();
//open the table
adcmd.SelectCommand =
       new OleDbCommand("Select * from " + tableName, conn);
OleDbCommandBuilder olecb = new OleDbCommandBuilder(adcmd);
adcmd.TableMappings.Add("Table", tableName);
//load current data into the local table copy
adcmd.Fill(dset, tableName);
//get the Enumerator from the Hashtable
IEnumerator ienum = names.Keys.GetEnumerator();
//move through the table, adding the names to new rows
while (ienum.MoveNext()) {
       string name = (string)ienum.Current;
       row = dtable.NewRow();     //get new rows
       row[columnName] = name;
       dtable.Rows.Add(row);    //add into table
}
//Now update the database with this table
try {
       adcmd.Update(dset);
       closeConn();
       filled = true;
}
catch (Exception e) {
       Console.WriteLine (e.Message);
}
```

It is this table editing and update process that is central to the ADO style of programming. You get the table, modify the table, and update the changes back to the database. You use this same process to edit or delete rows, and updating the database makes these changes as well.

Building the Façade Classes

This description is the beginning of the new Façade we are developing to handle creating, connecting to, and using databases. In order to carry out the rest, let's consider Table 18-4, grocery prices at three local stores.

Copyright © , 2002 by James W Cooper

```
Stop and Shop,      Apples,        0.27
Stop and Shop,      Oranges,       0.36
Stop and Shop,      Hamburger,     1.98
Stop and Shop,      Butter,        2.39
Stop and Shop,      Milk,          1.98
Stop and Shop,      Cola,          2.65
Stop and Shop,      Green beans,   2.29
Village Market,     Apples,        0.29
Village Market,     Oranges,       0.29
Village Market,     Hamburger,     2.45
Village Market,     Butter,        2.99
Village Market,     Milk,          1.79
Village Market,     Cola,          3.79
Village Market,     Green beans,   2.19
Waldbaum's,         Apples,        0.33
Waldbaum's,         Oranges,       0.47
Waldbaum's,         Hamburger,     2.29
Waldbaum's,         Butter,        3.29
Waldbaum's,         Milk,          1.89
Waldbaum's,         Cola,          2.99
Waldbaum's,         Green beans,   1.99
```

Table 18-4- Grocery Pricing Data

It would be nice if we had this information in a database so we could easily answer the question “Which store has the lowest prices for oranges?” Such a database should contain three tables: the supermarkets, the foods, and the prices. We also need to keep the relations among the three tables. One simple way to handle this is to create a Stores table with StoreName and StoreKey, a Foods table with a FoodName and a FoodKey, and a Price table with a PriceKey, a Price, and references to the StoreKey and Foodkey.

In our Façade, we will make each of these three tables its own class and have it take care of creating the actual tables. Since these three tables are so similar, we’ll derive them all from the basic DBTable class.

Copyright © , 2002 by James W Cooper

Building the Price Query

For every food name, we'd like to get a report of which stores have the cheapest prices. This means writing a simple SQL query against the database. We can do this within the Price class and have it return a Dataset with the store names and prices.

The final application simply fills one list box with the food names and files the other list box with prices when you click on a food name, as shown in Figure 18-1.

Figure 18-1 – The grocery program using a Façade pattern

Making the ADO.NET Façade

In the Façade we will make for our grocery database, we start with an abstract DBase class that represents a connection to a database. This encapsulates making the connection and opening a table and an SQL query.

```
public abstract class DBase      {
      protected OleDbConnection conn;

private void openConnection() {
      if (conn.State == ConnectionState.Closed){
```

Copyright © , 2002 by James W Cooper

```
            conn.Open ();
      }
}
//------
private void closeConnection() {
      if (conn.State == ConnectionState.Open ){
            conn.Close ();
      }
}
//------
public DataTable openTable (string tableName) {
      OleDbDataAdapter adapter = new OleDbDataAdapter ();
      DataTable dtable = null;
      string query = "Select * from " + tableName;
      adapter.SelectCommand = new OleDbCommand (query, conn);
      DataSet dset = new DataSet ("mydata");
      try {
            openConnection();
            adapter.Fill (dset);
            dtable = dset.Tables [0];
      }
      catch(Exception e) {
            Console.WriteLine (e.Message );
      }

      return dtable;
}
//------
public DataTable openQuery(string query) {
      OleDbDataAdapter dsCmd = new OleDbDataAdapter ();
      DataSet dset = new DataSet ();    //create a dataset
      DataTable dtable = null;          //declare a data table
      try {
            //create the command
            dsCmd.SelectCommand = new OleDbCommand(query, conn);
            openConnection();   //open the connection
            //fill the dataset
            dsCmd.Fill(dset, "mine");
            //get the table
            dtable = dset.Tables[0];
            closeConnection();               //always close it
            return dtable;                   //and return it
      }
      catch (Exception e) {
            Console.WriteLine (e.Message);
```

Copyright © , 2002 by James W Cooper

```
            return null;
      }
}
//------
public void openConnection(string connectionString) {
      conn = new OleDbConnection(connectionString);
}
//------
public OleDbConnection getConnection() {
      return conn;
}
}
```

Note that this class is complete except for constructors. We'll make derived classes that create the connection strings for various databases. We'll make a version for Access:

```
public class AxsDatabase :Dbase  {
      public AxsDatabase(string dbName)       {
            string connectionString =
            "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" +
                  dbName;
            openConnection(connectionString);
      }
}
```

and another for SQL Server.

```
public class SQLServerDatabase:DBase    {
      string connectionString;
      //-----
public SQLServerDatabase(String dbName)              {
       connectionString = "Persist Security Info = False;" +
            "Initial Catalog =" + dbName + ";" +
            "Data Source = myDataServer;User ID = myName;" +
            "password=";
      openConnection(connectionString);
}
//-----
public SQLServerDatabase(string dbName, string serverName,
      string userid, string pwd) {
      connectionString = "Persist Security Info = False;" +
            "Initial Catalog =" + dbName + ";" +
            "Data Source =" + serverName + ";" +
            "User ID =" + userid + ";" +
            "password=" + pwd;
```

Copyright © , 2002 by James W Cooper

```
      openConnection(connectionString);
     }
}
```

The DBTable class

The other major class we will need is the DBTable class. It encapsulates opening, loading, and updating a single database table. We will also use this class in this example to add the single values. Then we can derive food and store classes that do this addition for each class.

```
public class DBTable      {
      protected DBase db;
      protected string tableName;
      private bool filled, opened;
      private DataTable dtable;
      private int rowIndex;
      private Hashtable names;
      private string columnName;
      private DataRow row;
      private OleDbConnection conn;
      private int index;
//-----
public DBTable(DBase datab, string tb_Name)          {
      db = datab;
      tableName = tb_Name;
      filled =false;
      opened = false;
      names = new Hashtable();
}
//-----
public void createTable() {
      try {
            dtable = new DataTable(tableName);
            dtable.Clear();
      }
      catch (Exception e) {
            Console.WriteLine (e.Message );
      }
}
//-----
public bool hasMoreElements() {
      if(opened)
            return (rowIndex < dtable.Rows.Count) ;
      else
```

Copyright © , 2002 by James W Cooper

```
            return false;
}
//-----
public int getKey(string nm, string keyname){

      DataRow row;
      int key;
  if(! filled)
            return (int)names[ nm];
  else {
      string query = "select * from " + tableName + " where " +
             columnName + "='" + nm + "'";
      dtable = db.openQuery(query);
      row = dtable.Rows[0];
      key = Convert.ToInt32 (row[keyname].ToString());
      return key;
  }
}
//-----
public virtual void makeTable(string cName) {
      //shown below
//-----
private void closeConn() {
 if( conn.State == ConnectionState.Open) {
             conn.Close();
 }
}
//-----
private void openConn() {
      if(conn.State == ConnectionState.Closed ) {
            conn.Open();
      }
}
//-----
public void openTable() {
      dtable = db.openTable(tableName);
      rowIndex = 0;
      if(dtable != null)
            opened = true;
      }
//-----
 public void delete() {
      //shown above
 }
}
```

Copyright © , 2002 by James W Cooper

Creating Classes for Each Table

We can derive the Store, Food, and Prices classes from DBTable and reuse much of the code. When we parse the input file, both the Store and Food classes will require that we create a table of unique names: store names in one class and food names in the other.

C# provides a very convenient way to create these classes using the Hashtable. A Hashtable is an unbounded array where each element is identified with a unique key. One way people use Hashtables is to add objects to the table with a short nickname as the key. Then you can fetch the object from the table by using its nickname. The objects need not be unique, but, of course, the keys must be unique.

The other place Hashtables are convenient is in making a list of unique names. If we make the names the keys and some other number the contents, then we can add names to the Hashtable and assure ourselves that each will be unique. For them to be unique, the Hashtable must treat attempts to add a duplicate key in a predictable way. For example, the Java Hashtable simply replaces a previous entry having that key with the new one. The C# implementation of the Hashtable, on the other hand, throws an exception when we try to add a nonunique key value.

Now bearing in mind that we want to accumulate the entire list of names before adding them into the database, we can use the following method to add names to a Hashtable and make sure they are unique.

```
public void addTableValue(string nm) {
//accumulates names in hash table
       try {
              names.Add(nm, index++);
       }
       catch (ArgumentException) {}
      //do not allow duplicate names to be added
}
```

Copyright © , 2002 by James W Cooper

Then, once we have added all the names, we can add each of them to the database table. Here we use the Enumerator property of the Hashtable to iterate though all the names we have entered in the list.

```
public virtual void makeTable(string cName) {
       columnName = cName;
       //stores current hash table values in data table
       DataSet dset = new DataSet(tableName);  //create dataset
       dtable = new DataTable(tableName);   //and a datatable
       dset.Tables.Add(dtable);             //add to collection
       conn = db.getConnection();
       openConn();                      //open the connection
       OleDbDataAdapter adcmd = new OleDbDataAdapter();
       //open the table
       adcmd.SelectCommand =
              new OleDbCommand("Select * from " + tableName, conn);
       OleDbCommandBuilder olecb = new OleDbCommandBuilder(adcmd);
       adcmd.TableMappings.Add("Table", tableName);
       //load current data into the local table copy
       adcmd.Fill(dset, tableName);
       //get the Enumerator from the Hashtable
       IEnumerator ienum = names.Keys.GetEnumerator();
       //move through the table, adding the names to new rows
       while (ienum.MoveNext()) {
              string name = (string)ienum.Current;
              row = dtable.NewRow();     //get new rows
              row[columnName] = name;
              dtable.Rows.Add(row);    //add into table
       }
       //Now update the database with this table
       try {
              adcmd.Update(dset);
              closeConn();
              filled = true;
       }
       catch (Exception e) {
              Console.WriteLine (e.Message);
       }
}
```

This simplifies our derived Stores table to just the following.

```
public class Stores :DBTable     {
       public Stores(DBase db):base(db, "Stores"){
       }
```

Copyright © , 2002 by James W Cooper

```
   //-----
   public void makeTable() {
              base.makeTable ("Storename");
        }
}
```

And it simplifies the Foods table to much the same thing.

```
public class Foods: DBTable       {
       public Foods(DBase db):base(db, "Foods"){
       }
       //-----
       public void makeTable() {
              base.makeTable ("Foodname");
       }
       //-----
       public string getValue() {
              return base.getValue ("FoodName");
       }
}
```

The getValue method allows us to enumerate the list of names of Stores or Foods, and we can put it in the base DBTable class.

```
public virtual string getValue(string cname) {
 //returns the next name in the table
 //assumes that openTable has already been called
       if (opened) {
              DataRow row = dtable.Rows[rowIndex++];
              return row[cname].ToString().Trim ();
       }
       else
              return "";
 }
```

Note that we make this method *virtual* so we can override it where needed.

Building the Price Table

The Price table is a little more complicated because it contains keys from the other two tables. When it is completed, it will look like Table 18-5.

Pricekey	Foodkey	StoreKey	Price
1	1	1	0.27

Copyright © , 2002 by James W Cooper

2	2	1	0.36
3	3	1	1.98
4	4	1	2.39
5	5	1	1.98
6	6	1	2.65
7	7	1	2.29
8	1	2	0.29
9	2	2	0.29
10	3	2	2.45
11	4	2	2.99
12	5	2	1.79
13	6	2	3.79
14	7	2	2.19
15	1	3	0.33
16	2	3	0.47
17	3	3	2.29
18	4	3	3.29
19	5	3	1.89
20	6	3	2.99
21	7	3	1.99

Table 18-5 – The Price Table in the Grocery Database

To create it, we have to reread the file, finding the store and food names, looking up their keys, and adding them to the Price table. The DBTable interface doesn't include this final method, but we can add additional specific methods to the Price class that are not part of that interface.

The Prices class stores a series of StoreFoodPrice objects in an ArrayList and then loads them all into the database at once. Note that we have overloaded the classes of DBTable to take arguments for the store and food key values as well as the price.

Each time we add a storekey, foodkey and price to the internal ArrayList table, we create an instance of the StoreFoodPrice object and store it.

```
public class StoreFoodPrice {
```

Copyright © , 2002 by James W Cooper

```
        private int storeKey, foodKey;
        private float foodPrice;
        //-----
        public StoreFoodPrice(int sKey, int fKey, float fPrice) {
               storeKey = sKey;
               foodKey = fKey;
               foodPrice = fPrice;
        }
        //-----
        public int getStore() {
               return storeKey;
        }
        //-----
        public int getFood() {
               return foodKey;
        }
        //-----
        public float getPrice() {
               return foodPrice;
        }
}
```

Then, when we have them all, we create the actual database table:

```
public class Prices : DBTable    {
        private ArrayList priceList;
        public Prices(DBase db) : base(db, "Prices")   {
               priceList = new ArrayList ();
        }
        //-----
        public void makeTable() {
        //stores current array list values in data table
        OleDbConnection adc = new OleDbConnection();

        DataSet dset = new DataSet(tableName);
        DataTable dtable = new DataTable(tableName);

        dset.Tables.Add(dtable);
        adc = db.getConnection();
        if (adc.State == ConnectionState.Closed)
               adc.Open();
        OleDbDataAdapter adcmd = new OleDbDataAdapter();

         //fill in price table
         adcmd.SelectCommand =
```

Copyright © , 2002 by James W Cooper

```
            new OleDbCommand("Select * from " + tableName, adc);
          OleDbCommandBuilder custCB = new
            OleDbCommandBuilder(adcmd);
          adcmd.TableMappings.Add("Table", tableName);
          adcmd.Fill(dset, tableName);
          IEnumerator ienum = priceList.GetEnumerator();
          //add new price entries
        while (ienum.MoveNext() ) {
            StoreFoodPrice fprice =
                  (StoreFoodPrice)ienum.Current;
            DataRow row = dtable.NewRow();
            row["foodkey"] = fprice.getFood();
            row["storekey"] = fprice.getStore();
            row["price"] = fprice.getPrice();
            dtable.Rows.Add(row);    //add to table
        }
          adcmd.Update(dset);      //send back to database
          adc.Close();
          }
        //-----
        public DataTable getPrices(string food) {
           string query=
            "SELECT Stores.StoreName, " +
            "Foods.Foodname, Prices.Price " +
            "FROM (Prices INNER JOIN Foods ON " +
            "Prices.Foodkey = Foods.Foodkey) " +
            "INNER JOIN Stores ON " +
            "Prices.StoreKey = Stores.StoreKey " +
            "WHERE(((Foods.Foodname) = \'" + food + "\')) " +
            "ORDER BY Prices.Price";
          return db.openQuery(query);
        }
        //-----
        public void addRow(int storeKey, int foodKey, float price)
            priceList.Add (
                  new StoreFoodPrice (storeKey,
                        foodKey, price));
        }
}
```

Loading the Database Tables

With all these classes derived, we can write a class to load the table from the data file. It reads the file once and builds the Store and Food database

Copyright © , 2002 by James W Cooper

tables. Then it reads the file again and looks up the store and food keys and adds them to the array list in the Price class. Finally, it creates the Price table.

```
public class DataLoader    {
       private csFile vfile;
       private Stores store;
       private Foods fods;
       private Prices price;
       private DBase db;
       //-----
       public DataLoader(DBase datab)    {
              db = datab;
              store = new Stores(db);
              fods = new Foods (db);
              price = new Prices(db);
       }
       //-----
       public void load(string dataFile) {
              string sline;
              int storekey, foodkey;
              StringTokenizer tok;
              //delete current table contents
              store.delete();
              fods.delete();
              price.delete();
              //now read in new ones
              vfile = new csFile(dataFile);
              vfile.OpenForRead();
              sline = vfile.readLine();
              while (sline != null){
                     tok = new StringTokenizer(sline, ",");
                     store.addTableValue(tok.nextToken()); //store
                     fods.addTableValue(tok.nextToken());  //food
                     sline = vfile.readLine();
              }
              vfile.close();
              //construct store and food tables
              store.makeTable();
              fods.makeTable();
              vfile.OpenForRead();
              sline = vfile.readLine();
       while (sline != null) {
              //get the gets and add to storefoodprice objects
              tok = new StringTokenizer(sline, ",");
```

Copyright © , 2002 by James W Cooper

```
            storekey = store.getKey(tok.nextToken(), "Storekey");
            foodkey = fods.getKey(tok.nextToken(), "Foodkey");
            price.addRow(storekey, foodkey,
                  Convert.ToSingle (tok.neXtToken()));
                  sline = vfile.readLine();
      }
       //add all to price table
      price.makeTable();
      vfile.close();
      }
}
```

The Final Application

The program loads a list of food prices into a list box on startup.

```
private void loadFoodTable() {
       Foods fods =new Foods(db);
       fods.openTable();
       while (fods.hasMoreElements()){
            lsFoods.Items.Add(fods.getValue());
      }
}
```

And it displays the prices of the selected food when you click on it.

```
private void lsFoods_SelectedIndexChanged(object sender,
            System.EventArgs e) {
      string food  = lsFoods.Text;
      DataTable dtable = prc.getPrices(food);

      lsPrices.Items.Clear();
      foreach (DataRow rw in dtable.Rows) {
          lsPrices.Items.Add(rw["StoreName"].ToString().Trim() +
            "\t" + rw["Price"].ToString());
      }
}
```

The final program is shown in Figure 18-2.

Copyright © , 2002 by James W Cooper

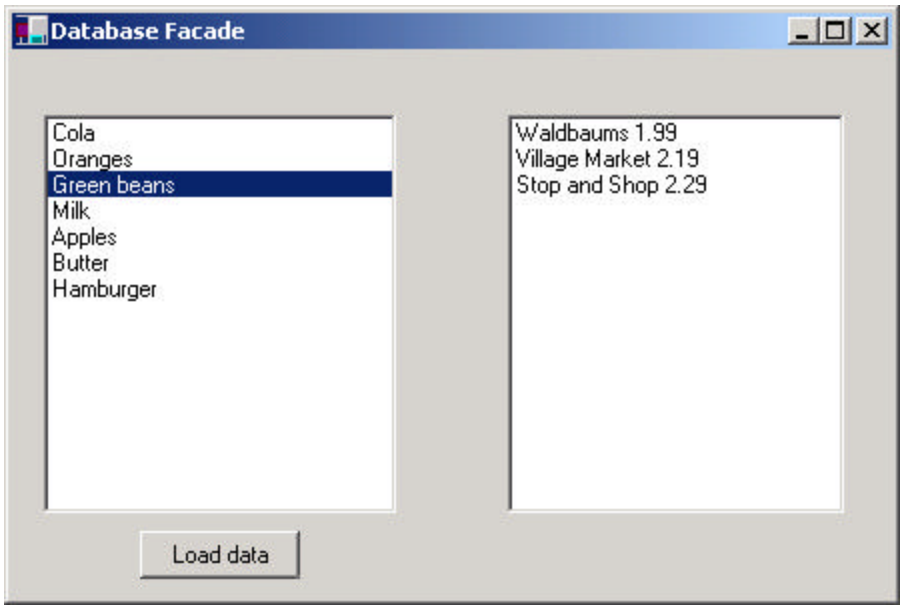


Figure 18-2– The C# grocery database program

If you click on the "load data" button, it clears the database and reloads it from the text file.

What Constitutes the Façade?

The Facade in this case wraps the classes as follows.

- Dbase

  —Contains ADOConnection, Database, DataTable, ADOCommand, ADODatasetCommand

- DBTable

  —Contains ADOCommand, Dataset, Datarow, Datatable, ADODatasetCommand

You can quickly see the advantage of the Façade approach when dealing with such complicated data objects.

Copyright © , 2002 by James W Cooper

Consequences of the Façade

The Façade pattern shields clients from complex subsystem components and provides a simpler programming interface for the general user. However, it does not prevent the advanced user from going to the deeper, more complex classes when necessary.

In addition, the Façade allows you to make changes in the underlying subsystems without requiring changes in the client code and reduces compilation dependencies.

Thought Question

Suppose you had written a program with a File|Open menu, a text field, and some buttons controlling font (bold and italic). Now suppose that you need to have this program run from a line command with arguments. Suggest how to use a Façade pattern.

Programs on the CD-ROM

`\Façade\`	C# database Façade classes

Copyright © , 2002 by James W Cooper

19. The Flyweight Pattern

The Flyweight pattern is used to avoid the overhead of large numbers of very similar classes. There are cases in programming where it seems that you need to generate a very large number of small class instances to represent data. Sometimes you can greatly reduce the number of different classes that you need to instantiate if you can recognize that the instances are fundamentally the same except for a few parameters. If you can move those variables outside the class instance and pass them in as part of a method call, the number of separate instances can be greatly reduced by sharing them.

The Flyweight design pattern provides an approach for handling such classes. It refers to the instance's *intrinsic* data that makes the instance unique and the *extrinsic* data that is passed in as arguments. The Flyweight is appropriate for small, fine-grained classes like individual characters or icons on the screen. For example, you might be drawing a series of icons on the screen in a window, where each represents a person or data file as a folder, as shown in Figure 19-1.

Copyright © , 2002 by James W Cooper

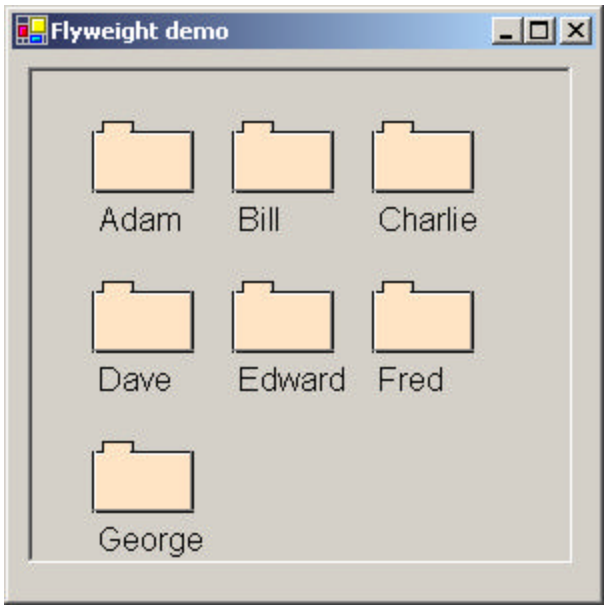


Figure 19-1– A set of folders representing information about various people. Since these are so similar, they are candidates for the Flyweight pattern.

In this case, it does not make sense to have an individual class instance for each folder that remembers the person's name and the icon's screen position. Typically, these icons are one of a few similar images, and the position where they are drawn is calculated dynamically based on the window's size in any case.

In another example in *Design Patterns*, each character in a document is represented as a single instance of a character class, but the positions where the characters are drawn on the screen are kept as external data, so there only has to be one instance of each character, rather than one for each appearance of that character.

Discussion

Flyweights are sharable instances of a class. It might at first seem that each class is a Singleton, but in fact there might be a small number of

Copyright © , 2002 by James W Cooper

instances, such as one for every character or one for every icon type. The number of instances that are allocated must be decided as the class instances are needed, and this is usually accomplished with a FlyweightFactory class. This Factory class usually *is* a Singleton, since it needs to keep track of whether a particular instance has been generated yet. It then either returns a new instance or a reference to one it has already generated.

To decide if some part of your program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it extrinsic. If this makes it possible to greatly reduce the number of different class instances your program needs to maintain, this might be a case where Flyweights will help.

Example Code

Suppose we want to draw a small folder icon with a name under it for each person in an organization. If this is a large organization, there could be a large number of such icons, but they are actually all the same graphical image. Even if we have two icons—one for "is Selected" and one for "not Selected"—the number of different icons is small. In such a system, having an icon object for each person, with its own coordinates, name, and selected state, is a waste of resources. We show two such icons in Figure 19-2.

Copyright © , 2002 by James W Cooper

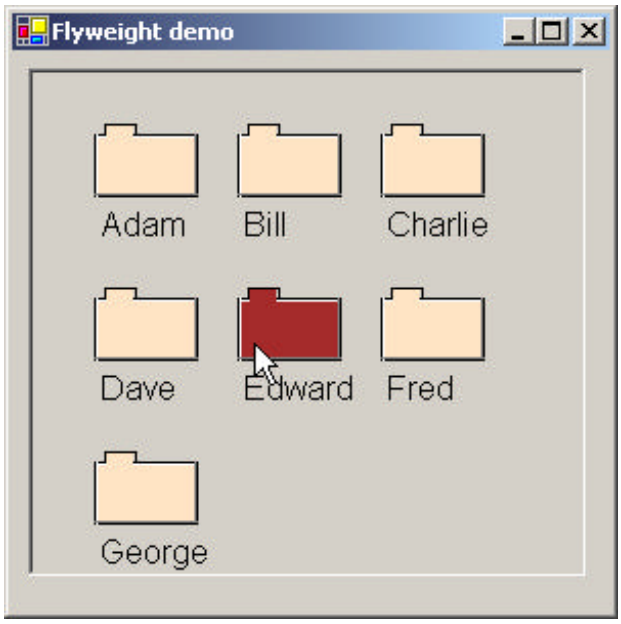


Figure 19-2– The Flyweight display with one folder selected

Instead, we'll create a FolderFactory that returns either the selected or the unselected folder drawing class but does not create additional instances once one of each has been created. Since this is such a simple case, we just create them both at the outset and then return one or the other.

```
public class FolderFactory {
       private Folder selFolder, unselFolder;
       //-----
       public FolderFactory()           {
              //create the two folders
              selFolder = new Folder(Color.Brown);
              unselFolder = new Folder(Color.Bisque);
       }
       //-----
       public Folder getFolder(bool selected) {
              if(selected)
                     return selFolder;
              else
                     return unselFolder;
```

Copyright © , 2002 by James W Cooper

```
        }
}
```

For cases where more instances could exist, the Factory could keep a table of those it had already created and only create new ones if they weren't already in the table.

The unique thing about using Flyweights, however, is that we pass the coordinates and the name to be drawn into the folder when we draw it. These coordinates are the extrinsic data that allow us to share the folder objects and, in this case, create only two instances. The complete folder class shown here simply creates a folder instance with one background color or the other and has a public draw method that draws the folder at the point you specify.

```
public class Folder        {
       //Draws a folder at the specified coordinates
       private const int w  = 50;
       private const int h = 30;
       private Pen blackPen, whitePen;
       private Pen grayPen;

       private SolidBrush backBrush, blackBrush;
       private Font fnt;
       //------
       public Folder(Color col)          {
              backBrush = new SolidBrush(col);
              blackBrush = new SolidBrush(Color.Black);
              blackPen = new Pen(Color.Black);
              whitePen = new Pen(Color.White);
              grayPen = new Pen(Color.Gray);
              fnt = new Font("Arial", 12);
       }
       //-----
       public void draw(Graphics g, int x, int y, string title) {
              //color folder
              g.FillRectangle(backBrush, x, y, w, h);
              //outline in black
              g.DrawRectangle(blackPen, x, y, w, h);
              //left 2 sides have white line
              g.DrawLine(whitePen, x + 1, y + 1, x + w - 1, y + 1);
              g.DrawLine(whitePen, x + 1, y, x + 1, y + h);
              //draw tab
```

Copyright © , 2002 by James W Cooper

```
            g.DrawRectangle(blackPen, x + 5, y - 5, 15, 5);
            g.FillRectangle(backBrush, x + 6, y - 4, 13, 6);
            //gray line on right and bottom
            g.DrawLine(grayPen, x, y + h - 1, x + w, y + h - 1);
            g.DrawLine(grayPen, x + w - 1, y, x + w - 1,
             y + h - 1);
            g.DrawString(title, fnt, blackBrush, x, y + h + 5);
      }
}
```

To use a Flyweight class like this, your main program must calculate the position of each folder as part of its paint routine and then pass the coordinates to the folder instance. This is actually rather common, since you need a different layout, depending on the window's dimensions, and you would not want to have to keep telling each instance where its new location is going to be. Instead, we compute it dynamically during the paint routine.

Here we note that we could have generated an ArrayList of folders at the outset and simply scan through the array to draw each folder. Such an array is not as wasteful as a series of different instances because it is actually an array of references to one of only two folder instances. However, since we want to display one folder as "selected," and we would like to be able to change which folder is selected dynamically, we just use the FolderFactory itself to give us the correct instance each time.

There are two places in our display routine where we need to compute the positions of folders: when we draw them, and when we check for a mouse hovering over them. Thus, it is convenient to abstract out the positioning code into a Positioner class:

```
public class Positioner    {
      private const int pLeft = 30;
      private const int pTop  = 30;
      private const int HSpace = 70;
      private const int VSpace = 80;
      private const int rowMax = 2;
      private int x, y, cnt;
      //-----
```

Copyright © , 2002 by James W Cooper

```
        public Positioner() {
                reset();
        }
        //-----
        public void reset() {
                x = pLeft;
                y = pTop;
                cnt = 0;
        }
        //-----
        public int nextX() {
                return x;
        }
        //-----
        public void incr() {
                cnt++;
                if (cnt > rowMax) { //reset to start new row
                        cnt = 0;
                        x = pLeft;
                        y += VSpace;
                }
                else {
                        x += HSpace;
                }
        }
        //-----
        public int nextY() {
                return y;
        }
}
```

Then we can write a much simpler paint routine:

```
private void picPaint(object sender,  PaintEventArgs e ) {
        Graphics g = e.Graphics;
        posn.reset ();
        for(int i = 0; i < names.Count; i++) {
                fol = folFact.getFolder(selectedName.Equals(
                        (string)names[i]));
                fol.draw(g, posn.nextX() , posn.nextY (),
                        (string)names[i]);
                posn.incr();
        }
}
```

Copyright © , 2002 by James W Cooper

The Class Diagram

The diagram in Figure 19-3 shows how these classes interact.

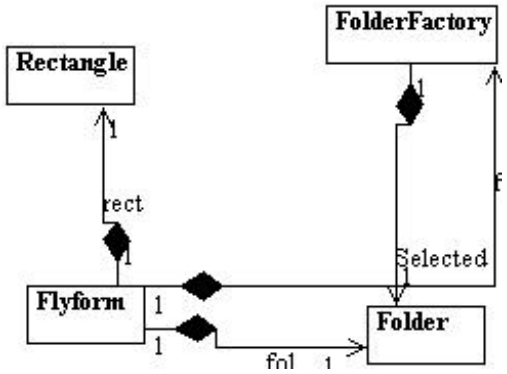


Figure 19-3 – How Flyweights are generated

The FlyCanvas class is the main UI class, where the folders are arranged and drawn. It contains one instance of the FolderFactory and one instance of the Folder class. The FolderFactory class contains two instances of Folder: *selected* and *unselected*. One or the other of these is returned to the FlyCanvas by the FolderFactory.

Selecting a Folder

Since we have two folder instances, selected and unselected, we'd like to be able to select folders by moving the mouse over them. In the previous paint routine, we simply remember the name of the folder that was selected and ask the factory to return a "selected' folder for it. Since the folders are not individual instances, we can't listen for mouse motion within each folder instance. In fact, even if we did listen within a folder, we'd need a way to tell the other instances to deselect themselves.

Instead, we check for mouse motion at the Picturebox level, and if the mouse is found to be within a Rectangle, we make that corresponding name the selected name. We create a single instance of a Rectangle class

Copyright © , 2002 by James W Cooper

where the testing can be done as to whether a folder contains the mouse at that instant. Note that we make this class part of the csPatterns namespace to make sure it does not collide with the Rectangle class in the System.Drawing namespace.

```
namespace csPatterns {
       public class Rectangle     {
       private int x1, x2, y1, y2;
       private int w, h;
       public Rectangle()         {                    }
       //-----
       public void init(int x, int y) {
              x1 = x;
              y1 = y;
              x2 = x1 + w;
              y2 = y1 + h;
       }
       //-----
       public void setSize(int w_, int h_) {
              w = w_;
              h = h_;
       }
       //-----
       public bool contains(int xp, int yp) {
              return (x1 <= xp) && (xp <= x2) &&
                     (y1 <= yp) && (yp <= y2);
       }
  }
}
```

This allows us to just check each name when we redraw and create a selected folder instance where it is needed.

```
private void Pic_MouseMove(object sender, MouseEventArgs e) {
       string oldname = selectedName;  //save old name
       bool found = false;
       posn.reset ();
       int i = 0;
       selectedName = "";
       while (i < names.Count && ! found) {
              rect.init (posn.nextX() , posn.nextY ());
              //see if a rectangle contains the mouse
              if (rect.contains(e.X, e.Y) ){
```

Copyright © , 2002 by James W Cooper

```
                    selectedName = (string)names[i];
                    found = true;
                }
                posn.incr ();
                i++;
            }
            //only refresh if mouse in new rectangle
            if(  !oldname.Equals ( selectedName)) {
                Pic.Refresh();
            }
}
```

Handling the Mouse and Paint Events

In C# we intercept the paint and mouse events by adding event handlers. To do the painting of the folders, we add a paint event handler to the picture box.

```
Pic.Paint += new PaintEventHandler (picPaint);
```

The picPaint handler we add draws the folders, as we showed above. We added this code manually because we knew the signature of a paint routine:

```
private void picPaint(object sender,  PaintEventArgs e ) {
```

While the mouse move event handler is very much analogous, we might not remember its exact form. So, we use the Visual Studio IDE to generate it for us. While displaying the form in design mode, we click on the PictureBox and in the Properties window we click on the lightning bolt to display the possible events for the PictureBox, as shown in Figure 19-4.

Copyright © , 2002 by James W Cooper

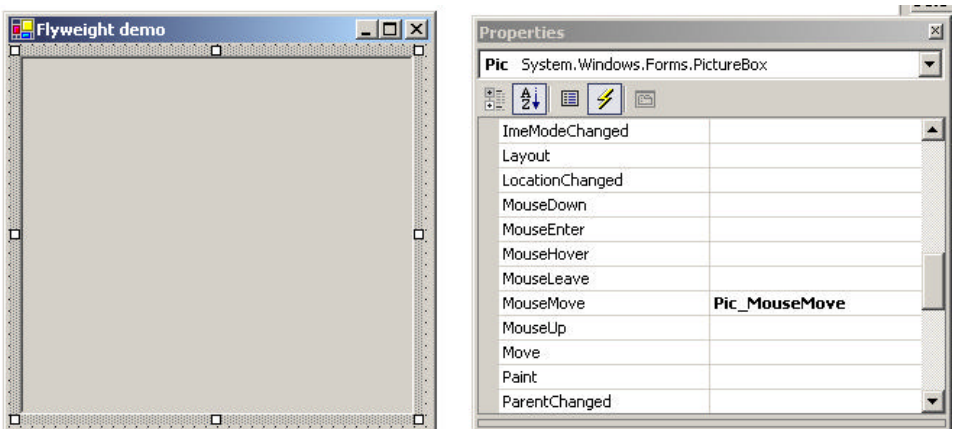


Figure 19-4 – Selecting the MouseMove event from the Properties window.

Then we double click on MouseMove, and it generates the correct code for the mouse move event and adds the event handler automatically. The generated empty method is just:

```
private void Pic_MouseMove(object sender, MouseEventArgs e) {
}
```

and the code generated to add the event handler is inside the Windows Form Designer generated section. It amounts to

```
Pic.MouseMove += new MouseEventHandler(Pic_MouseMove);
```

Flyweight Uses in C#

Flyweights are not frequently used at the application level in C#. They are more of a system resource management technique used at a lower level than C#. However, there are a number of stateless objects that get created in Internet programming that are somewhat analogous to Flyweights. It is generally useful to recognize that this technique exists so you can use it if you need it.

Some objects within the C# language could be implemented under the covers as Flyweights. For example, if there are two instances of a String

Copyright © , 2002 by James W Cooper

constant with identical characters, they could refer to the same storage location. Similarly, it might be that two integer or float constants that contain the same value could be implemented as Flyweights, although they probably are not.

Sharable Objects

The *Smalltalk Companion* points out that sharable objects are much like Flyweights, although the purpose is somewhat different. When you have a very large object containing a lot of complex data, such as tables or bitmaps, you would want to minimize the number of instances of that object. Instead, in such cases, you'd return one instance to every part of the program that asked for it and avoid creating other instances.

A problem with such sharable objects occurs when one part of a program wants to change some data in a shared object. You then must decide whether to change the object for all users, prevent any change, or create a new instance with the changed data. If you change the object for every instance, you may have to notify them that the object has changed.

Sharable objects are also useful when you are referring to large data systems outside of C#, such as databases. The DBase class we developed previously in the Façade pattern could be a candidate for a sharable object. We might not want a number of separate connections to the database from different program modules, preferring that only one be instantiated. However, should several modules in different threads decide to make queries simultaneously, the Database class might have to queue the queries or spawn extra connections.

Copy-on-Write Objects

The Flyweight pattern uses just a few object instances to represent many different objects in a program. All of them normally have the same base properties as intrinsic data and a few properties that represent extrinsic data that vary with each manifestation of the class instance. However, it could occur that some of these instances eventually take on new intrinsic

Copyright © , 2002 by James W Cooper

properties (such as shape or folder tab position) and require a new specific instance of the class to represent them. Rather than creating these in advance as special subclasses, it is possible to copy the class instance and change its intrinsic properties when the program flow indicates that a new separate instance is required. The class copies this itself when the change becomes inevitable, changing those intrinsic properties in the new class. We call this process "copy-on-write" and can build this into Flyweights as well as a number of other classes, such as the Proxy, which we discuss next.

Thought Question

If Buttons can appear on several different tabs of a TabDialog, but each of them controls the same one or two tasks, is this an appropriate use for a Flyweight?

Programs on the CD-ROM

`\Flyweight`	C# folders

Copyright © , 2002 by James W Cooper

20. The Proxy Pattern

The Proxy pattern is used when you need to represent an object that is complex or time consuming to create, by a simpler one. If creating an object is expensive in time or computer resources, Proxy allows you to postpone this creation until you need the actual object. A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

There are several cases where a Proxy can be useful.

1. An object, such as a large image, takes a long time to load.
2. The results of a computation take a long time to complete, and you need to display intermediate results while the computation continues.
3. The object is on a remote machine, and loading it over the network may be slow, especially during peak network load periods.
4. The object has limited access rights, and the proxy can validate the access permissions for that user.

Proxies can also be used to distinguish between requesting an instance of an object and the actual need to access it. For example, program initialization may set up a number of objects that may not all be used right away. In that case, the proxy can load the real object only when it is needed.

Let's consider the case of a large image that a program needs to load and display. When the program starts, there must be some indication that an image is to be displayed so that the screen lays out correctly, but the actual image display can be postponed until the image is completely loaded. This is particularly important in programs such as word processors and Web browsers that lay out text around the images even before the images are available.

Copyright © , 2002 by James W Cooper

An image proxy can note the image and begin loading it in the background while drawing a simple rectangle or other symbol to represent the image's extent on the screen before it appears. The proxy can even delay loading the image at all until it receives a paint request and only then begin the process.

Sample Code

In this example, we create a simple program to display an image on a Image control when it is loaded. Rather than loading the image directly, we use a class we call ImageProxy to defer loading and draw a rectangle until loading is completed.

```
private void init() {
       imgProxy = new ImageProxy ();
}
//-----
public Form1() {
       InitializeComponent();
       init();
}
//-----
private void button1_Click(object sender, EventArgs e) {
       Pic.Image = imgProxy.getImage ();
}
```

Note that we create the instance of the ImageProxy just as we would have for an Image. The ImageProxy class sets up the image loading and creates an Imager object to follow the loading process. It returns a class that implements the Imager interface.

```
public interface Imager    {
       Image getImage() ;
}
```

In this simple case, the ImageProxy class just delays five seconds and then switches from the preliminary image to the final image. It does this using

Copyright © , 2002 by James W Cooper

an instance of the Timer class. Timers are handled using a TimerCallback class that defines the method to be called when the timer ticks. This is much the same as the way we add other event handlers. And this callback method *timerCall* sets the done flag and turns off the timer.

```
public class ImageProxy    {
       private bool done;
       private Timer timer;
       //-----
       public ImageProxy()               {
       //create a timer thread and start it
              timer = new Timer (
                 new TimerCallback (timerCall), this, 5000, 0);
       }
       //-----
       //called when timer completes
       private void timerCall(object obj) {
              done = true;
              timer.Dispose ();
       }
       //-----
       public Image getImage() {
              Imager img;
              if (done)
                     img = new FinalImage ();
              else
                     img = new QuickImage ();
              return img.getImage ();
       }
}
```

We implement the Imager interface in two tiny classes we called QuickImage and FinalImage. One gets a small gif image and the other a larger (and presumably slower) jpeg image. In C#, Image is an abstract class, and the Bitmap, Cursor, Icon, and Metafile classes are derived from it. So the actual class we will return is derived from Image. The QuickImage class returns a Bitmap from a gif file, and the final image a JPEG file.

```
public class QuickImage : Imager  {
       public QuickImage() {}
       public Image getImage() {
```

Copyright © , 2002 by James W Cooper

```
                return new Bitmap ("Box.gif");
        }
}
//------------
public class FinalImage :Imager {
        public FinalImage() {}
        public Image getImage() {
                return new Bitmap("flowrtree.jpg");
        }
}
```

When you go to fetch an image, you initially get the quick image, and after five seconds, if you call the method again, you get the final image. The program's two states are illustrated in Figure 20-1

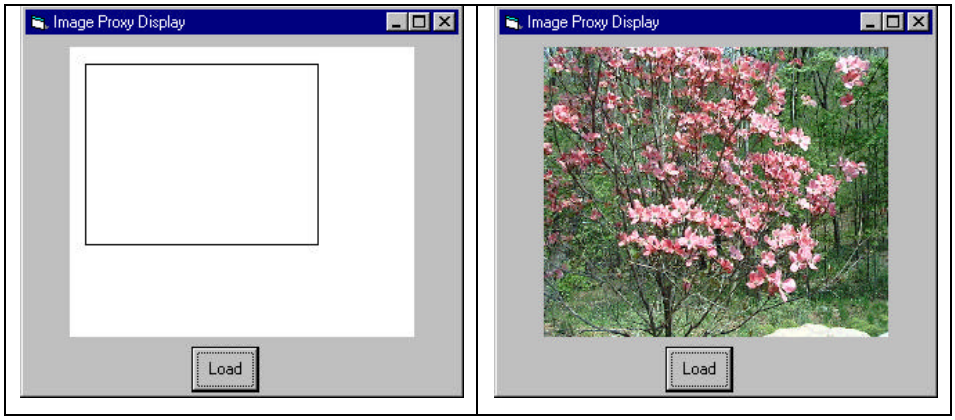


Figure 20-1 – The proxy image display on the left is shown until the image loads as shown on the right.

Proxies in C#

You see more proxy-like behavior in C# than in other languages, because it is crafted for network and Internet use. For example, the ADO.Net database connection classes are all effectively proxies.

Copyright © , 2002 by James W Cooper

Copy-on-Write

You can also use proxies is to keep copies of large objects that may or may not change. If you create a second instance of an expensive object, a Proxy can decide there is no reason to make a copy yet. It simply uses the original object. Then, if the program makes a change in the new copy, the Proxy can copy the original object and make the change in the new instance. This can be a great time and space saver when objects do not always change after they are instantiated.

Comparison with Related Patterns

Both the Adapter and the Proxy constitute a thin layer around an object. However, the Adapter provides a different interface for an object, while the Proxy provides the same interface for the object but interposes itself where it can postpone processing or data transmission effort.

A Decorator also has the same interface as the object it surrounds, but its purpose is to add additional (sometimes visual) function to the original object. A proxy, by contrast, controls access to the contained class.

Thought Question

You have designed a server that connects to a database. If several clients connect to your server at once, how might Proxies be of help?

Programs on the CD-ROM

`\Proxy`	Image proxy

Copyright © , 2002 by James W Cooper

Summary of Structural Patterns

Part 3 covered the following structural patterns.

The **Adapter** pattern is used to change the interface of one class to that of another one.

The **Bridge** pattern is designed to separate a class's interface from its implementation so you can vary or replace the implementation without changing the client code.

The **Composite** pattern is a collection of objects, any one of which may be either itself a Composite or just a leaf object.

The **Decorator** pattern, a class that surrounds a given class, adds new capabilities to it and passes all the unchanged methods to the underlying class.

The **Façade** pattern groups a complex set of objects and provides a new, simpler interface to access those data.

The **Flyweight** pattern provides a way to limit the proliferation of small, similar instances by moving some of the class data outside the class and passing it in during various execution methods.

The **Proxy** pattern provides a simple placeholder object for a more complex object that is in some way time consuming or expensive to instantiate

Copyright © , 2002 by James W Cooper

Part 4. Behavioral Patterns

Behavioral patterns are most specifically concerned with communication between objects. In Part 4, we examine the following.

The **Chain of Responsibility** allows a decoupling between objects by passing a request from one object to the next in a chain until the request is recognized.

The **Command pattern** utilizes simple objects to represent execution of software commands and allows you to support logging and undoable operations.

The **Interpreter pattern** provides a definition of how to include language elements in a program.

The **Iterator pattern** formalizes the way we move through a list of data within a class.

The **Mediator pattern** defines how communication between objects can be simplified by using a separate object to keep all objects from having to know about each other.

The **Memento pattern** defines how you might save the contents of an instance of a class and restore it later.

The **Observer pattern** defines the way a number of objects can be notified of a change,

The **State pattern** allows an object to modify its behavior when its internal state changes.

The **Strategy pattern** encapsulates an algorithm inside a class.

The **Template Method pattern** provides an abstract definition of an algorithm.

The **Visitor pattern** adds polymorphic functions to a class noninvasively.

Copyright © , 2002 by James W Cooper

21. Chain of Responsibility

The Chain of Responsibility pattern allows a number of classes to attempt to handle a request without any of them knowing about the capabilities of the other classes. It provides a loose coupling between these classes; the only common link is the request that is passed between them. The request is passed along until one of the classes can handle it.

One example of such a chain pattern is a Help system like the one shown in Figure 21-1. This is a simple application where different kinds of help could be useful, where every screen region of an application invites you to seek help but in which there are window background areas where more generic help is the only suitable result.

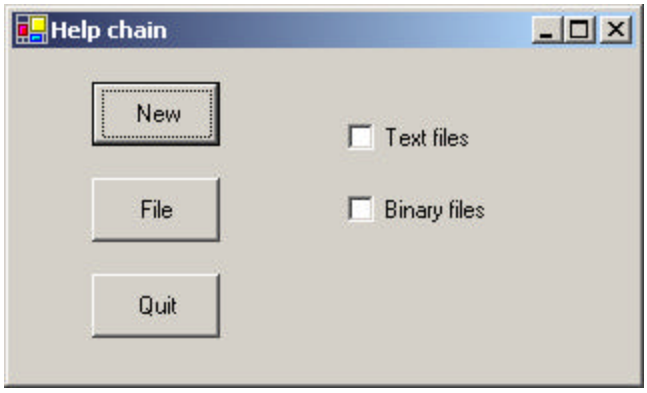


Figure 21-1 – A simple application where different kinds of help could be useful

When you select an area for help, that visual control forwards its ID or name to the chain. Suppose you selected the "New" button. If the first module can handle the New button, it displays the help message. If not, it forwards the request to the next module. Eventually, the message is forwarded to an "All buttons" class that can display a general message about how buttons work. If there is no general button help, the message is forwarded to the general help module that tells you how the system works

Copyright © , 2002 by James W Cooper

in general. If that doesn't exist, the message is lost, and no information is displayed. This is illustrated in Figure 21-2

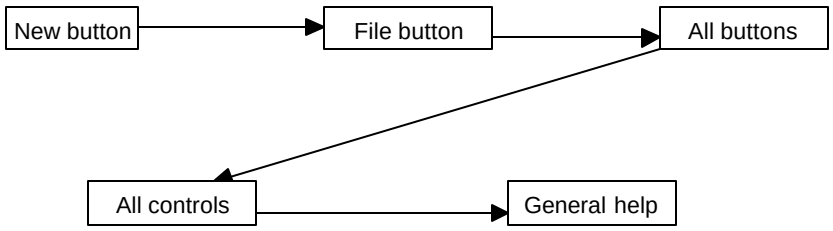


Figure 21-2– A simple Chain of Responsibility

There are two significant points we can observe from this example: first, the chain is organized from most specific to most general, and second, there is no guarantee that the request will produce a response in all cases. We will see shortly that you can use the Observer pattern to provide a way for a number of classes to be notified of a change,

Applicability

The Chain of Responsibility is a good example of a pattern that helps keep knowledge separate of what each object in a program can do. In other words, it reduces the coupling between objects so that they can act independently. This also applies to the object that constitutes the main program and contains instances of the other objects. You will find this pattern helpful in the following situations.

- There are several objects with similar methods that could be appropriate for the action the program is requesting. However, it is more appropriate for the objects to decide which one is to carry out the action than it is for you to build this decision into the calling code.

Copyright © , 2002 by James W Cooper

- One of the objects may be most suitable, but you don't want to build in a series of if-else or switch statements to select a particular object.
- There might be new objects that you want to add to the possible list of processing options while the program is executing.
- There might be cases when more than one object will have to act on a request, and you don't want to build knowledge of these interactions into the calling program.

Sample Code

The help system we just described is a little involved for a first example. Instead, let's start with a simple visual command-interpreter program (Figure 21-3) that illustrates how the chain works. This program displays the results of typed-in commands. While this first case is constrained to keep the example code tractable, we'll see that this Chain of Responsibility pattern is commonly used for parsers and even compilers.

In this example, the commands can be any of the following.

- Image filenames
- General filenames
- Color names
- All other commands

In the first three cases, we can display a concrete result of the request, and in the fourth case, we can only display the request text itself.

Copyright © , 2002 by James W Cooper

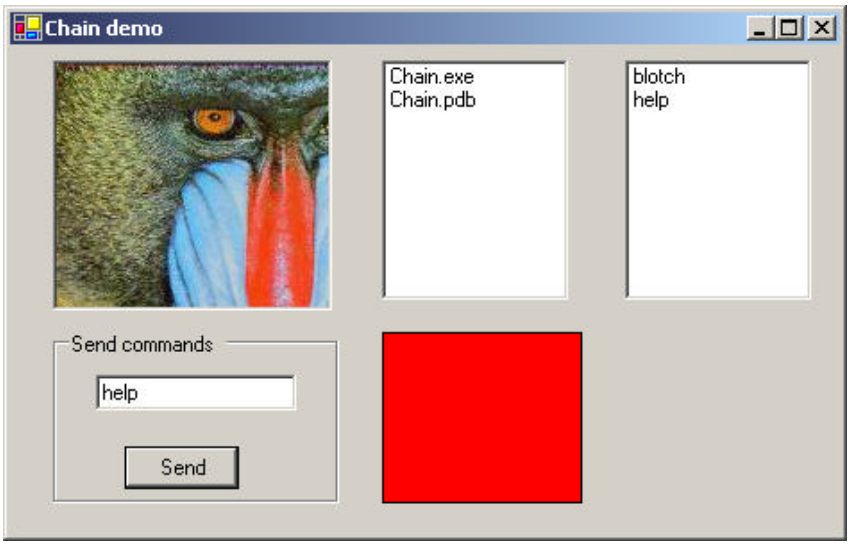
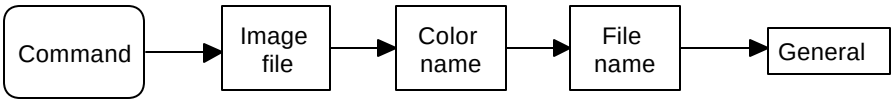


Figure 21-3– A simple visual command interpreter program that acts on one of four panels, depending on the command you type in.

In the preceding example system, we do the following.

1. We type in “Mandrill” and see a display of the image Mandrill.jpg.
2. Then we type in “File,” and that filename is displayed in the center list box.
3. Next, we type in “blue,” and that color is displayed in the lower center panel.

Finally, if we type in anything that is neither a filename nor a color, that text is displayed in the final, right-hand list box. This is shown in Figure 22-4.



Copyright © , 2002 by James W Cooper

Figure 21-4 – How the command chain works for the program in Figure 20-3

To write this simple chain of responsibility program, we start with an abstract Chain class.

```
public abstract class Chain      {
       //describes how all chains work
       private bool hasLink;
       protected Chain chn;
       public Chain()      {
              hasLink = false;
       }
       //you must implement this in derived classes
       public abstract void sendToChain(string mesg);
       //-----
       public void addToChain(Chain c) {
              //add new element to chain
              chn = c;
              hasLink = true;     //flag existence
       }
       //-----
       public Chain getChain() {
              return chn;         //get the chain link
       }
       //-----
       public bool hasChain() {
              return hasLink;     //true if linked to another
       }
       //-----
       protected void sendChain(string mesg) {
              //send message on down the chain
              if(chn != null)
                     chn.sendToChain (mesg);
       }
}
```

The *addChain* method adds another class to the chain of classes. The *getChain* method returns the current class to which messages are being forwarded. These two methods allow us to modify the chain dynamically and add additional classes in the middle of an existing chain. The *sendToChain* method forwards a message to the next object in the chain. And the protected sendChain method only sends the message down the chain if the next link is not null.

Copyright © , 2002 by James W Cooper

Our main program assembles the Chain classes and sets a reference to a control into each of them. We start with the ImageChain class, which takes the message string and looks for a .jpg file of that name. If it finds one, it displays it in the Image control, and if not, it sends the command on to the next element in the chain.

```
public class ImageChain :Chain    {
       PictureBox picBox;         //image goes here
       //-----
       public ImageChain(PictureBox pc)        {
              picBox = pc;        //save reference
       }
       //-----
       public override void sendToChain(string mesg) {
              //put image in picture box
              string fname = mesg + ".jpg";
              //assume jpg filename
              csFile fl = new csFile(fname);
              if(fl.exists())
                     picBox.Image = new Bitmap(fname);
              else{
                     if (hasChain()){    //send off down chain
                            chn.sendToChain(mesg);
                     }
              }

       }
}
```

In a similar fashion, the ColorChain class simply interprets the message as a color name and displays it if it can. This example only interprets three colors, but you could implement any number. Note how we interpret the color names by using them as keys to a Hashtable of color objects whee the string names are thekeys.

```
public class ColorChain : Chain {
       private Hashtable colHash; //color list kept here
       private Panel panel;              //color goes here
       //-----
       public ColorChain(Panel pnl)            {
              panel = pnl;               //save reference
              //create Hash table to correlate color names
```

Copyright © , 2002 by James W Cooper

```
                //with actual Color objects
                colHash = new Hashtable ();
                colHash.Add ("red", Color.Red);
                colHash.Add ("green", Color.Green);
                colHash.Add ("blue", Color.Blue);
        }
        //-----
        public override void sendToChain(string mesg) {
                mesg = mesg.ToLower ();
                try {
                        Color c = (Color)colHash[mesg];
                        //if this is a color, put it in the panel
                        panel.BackColor =c;
                }
                catch (NullReferenceException e) {
                        //send on if this doesn't work
                        sendChain(mesg);
                }

        }
}
```

The List Boxes

Both the file list and the list of unrecognized commands are ListBoxes. If the message matches part of a filename, the filename is displayed in the fileList box, and if not, the message is sent on to the NoComd chain element.

```
public override void sendToChain( string mesg) {
        //if the string matches any part of a filename
        //put those filenames in the file list box
        string[] files;
        string fname = mesg + "*.*";
        files = Directory.GetFiles(
                Directory.GetCurrentDirectory(), fname);
        //add them all to the listbox
        if (files.Length > 0){
                for (int i = 0; i< files.Length; i++) {
                        csFile vbf = new csFile(files[i]);
                        flist.Items.Add(vbf.getRootName());
                }
        }
        else {
```

Copyright © , 2002 by James W Cooper

```
            if ( hasChain()) {
                  chn.sendToChain(mesg);
            }
      }
}
```

The NoCmd Chain class is very similar. It, however, has no class to which to send data.

```
public class NoCmd :Chain  {
      private ListBox lsNocmd;   //commands go here
      //-----
      public NoCmd(ListBox lb)         {
            lsNocmd = lb;              //copy reference
      }
      //-----
      public override void sendToChain(string mesg) {
            //adds unknown commands to list box
            lsNocmd.Items.Add (mesg);
      }
}
```

Finally, we link these classes together in the Form_Load routine to create the Chain.

```
private void init() {
      //set up chains
      ColorChain clrChain = new ColorChain(pnlColor);
      FileChain flChain = new FileChain(lsFiles);
      NoCmd noChain = new NoCmd(lsNocmd);
      //create chain links
      chn = new ImageChain(picImage);
      chn.addToChain(clrChain);
      clrChain.addToChain(flChain);
      flChain.addToChain(noChain);
}
```

Finally, we kick off the chain by clicking on the Send button, which takes the current message in the text box and sends it along the chain.

```
private void btSend_Click(object sender, EventArgs e) {
                  chn.sendToChain (txCommand.Text );
```

Copyright © , 2002 by James W Cooper

```
}
```

You can see the relationship between these classes in the UML diagram in Figure 21-5.

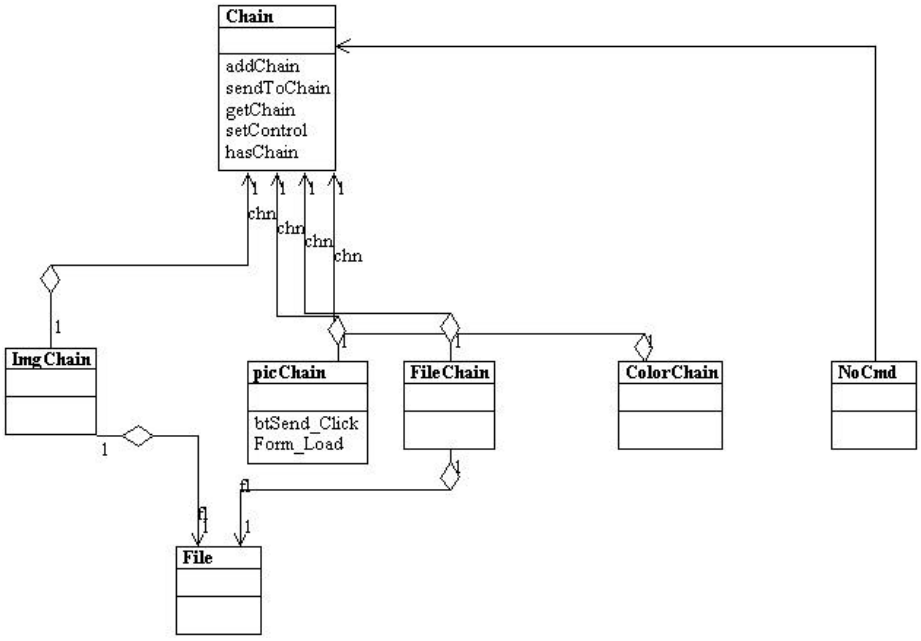


Figure 21-5– The class strcuture of the Chain of Responsibility program

The Sender class is the initial class that implements the Chain interface. It receives the button clicks and obtains the text from the text field. It passes the command on to the Imager class, the FileList class, the ColorImage class, and finally to the NoCmd class.

Programming a Help System

As we noted at the beginning of this discussion, help systems provide good examples of how the Chain of Responsibility pattern can be used. Now that we've outlined a way to write such chains, we'll consider a help

Copyright © , 2002 by James W Cooper

system for a window with several controls. The program (Figure 21-6) pops up a help dialog message when the user presses the F1 (help) key. The message depends on which control is selected when the F1 key is pressed.

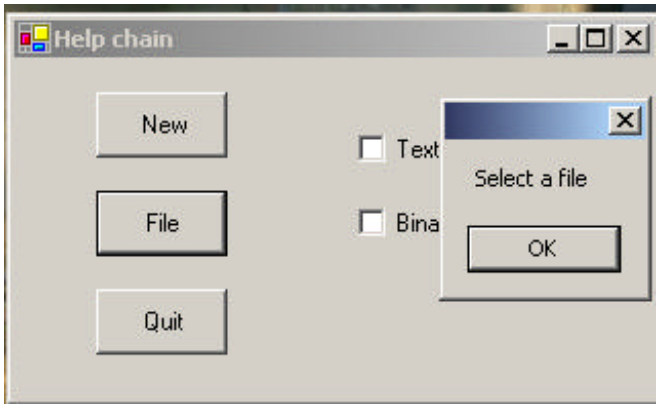


Figure 21-6 – A simple help demonstration

In the preceding example, the user has selected the Quit key, which does not have a specific help message associated with it. Instead, the chain forwards the help request to a general button help object that displays the message shown on the right.

To write this help chain system, we begin with an abstract Chain class that has handles Controls instead of messages. Note that no message is passed into the sendToChain method, and that the current control is stored in the class.

```
public abstract class Chain      {
       //describes how all chains work
       private bool hasLink;
       protected Control control;
       protected Chain chn;
       protected string message;

       public Chain(Control c, string mesg)    {
              hasLink = false;
              control = c;        //save the control
```

Copyright © , 2002 by James W Cooper

```
            message = mesg;
        }

        public abstract void sendToChain();
        //-----
        public void addToChain(Chain c) {
            //add new element to chain
            chn = c;
            hasLink = true;            //flag existence
        }
        //-----
        public Chain getChain() {
            return chn;   //get the chain link
        }
        //-----
        public bool hasChain() {
            return hasLink;            //true if linked to nother
            }
        //-----
        protected void sendChain() {
        //send message on down the chain
            if(chn != null)
                  chn.sendToChain ();
        }
}
```

Then you might create specific classes for each of the help message categories you want to produce. As we illustrated earlier, we want help messages for the following.

- The New button
- The File button
- A general button
- A general visual control (covering the check boxes)

In C#, one control will always have the focus, and thus we don't really need a case for the Window itself. However, we'll include one for completeness. However, there is little to be gained by creating separate classes for each message and assigning different controls to them. Instead, we'll create a general ControlChain class and pass in the control and the

Copyright © , 2002 by James W Cooper

message. Then, within the class it checks to see if that control has the focus, and if it does, it issues the associated help message:

```
public class ControlChain:Chain  {
       public ControlChain(Control c, string mesg):base(c, mesg)
       {}
       public override void sendToChain() {
              //if it has the focus display the message
              if (control.Focused ) {
                     MessageBox.Show (message);
              }
              else
                     //otherwise pass on down the chain
                     sendChain();
       }
}
```

Finally, we need one special case: the end of chain which will display a message regardless of whether the control has the focus. This is the EndChain class, and it is for completeness. Since one of the controls will presumably always have the focus, it is unlikely ever to be called:

```
public class EndChain:Chain      {
       public EndChain(Control c, string mesg):base(c, mesg){}
       //default message display class
       public override void sendToChain() {
              MessageBox.Show (message);
       }
}
```

We construct the chain in the form initializer as follows:

```
chn = new ControlChain(btNew, "Create new files");
Chain fl =new ControlChain (btFile, "Select a file");
chn.addToChain (fl);
Chain bq = new ControlChain (btQuit, "Exit from program");
fl.addToChain (bq);
Chain cb =new ControlChain (ckBinary, "Use binary files");
bq.addToChain (cb);
Chain ct =  new ControlChain (ckText, "Use text files");
cb.addToChain (ct);
Chain ce = new EndChain (this, "General message");
ct.addToChain (ce);
```

Copyright © , 2002 by James W Cooper

Receiving the Help Command

Now we need to assign keyboard listeners to look for the F1 keypress. At first, you might think we need five such listeners—for the three buttons and the two check boxes. However, we can simply make a single KeyDown event listener and assign it to each of the controls:

```
KeyEventHandler keyev =  new KeyEventHandler(Form1_KeyDown);
       btNew.KeyDown += keyev;
       btFile.KeyDown += keyev;
       btQuit.KeyDown += keyev;
       ckBinary.KeyDown += keyev;
       ckText.KeyDown += keyev;
```

Then, of course the key-down event launches the chain if the F1 key is pressed:

```
private void Form1_KeyDown(object sender, KeyEventArgs e) {
       if(e.KeyCode  == Keys.F1 )
              chn.sendToChain ();
}
```

We show the complete class diagram for this help system in Figure 21-7.

Copyright © , 2002 by James W Cooper

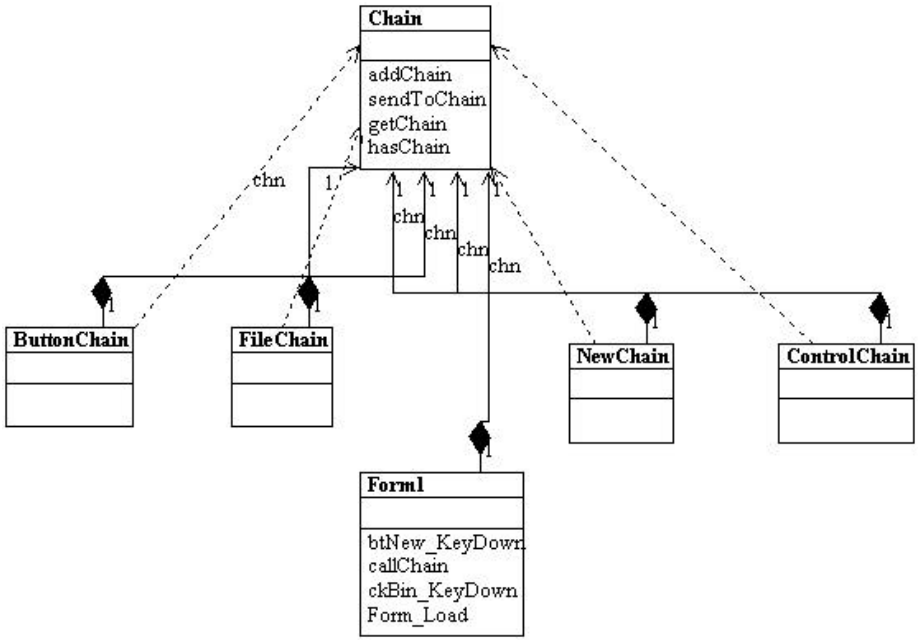


Figure 21-7 – The class diagram for the Help system

A Chain or a Tree?

Of course, a Chain of Responsibility does not have to be linear. The *Smalltalk Companion* suggests that it is more generally a tree structure with a number of specific entry points all pointing upward to the most general node, as shown in Figure 21-8..

Copyright © , 2002 by James W Cooper

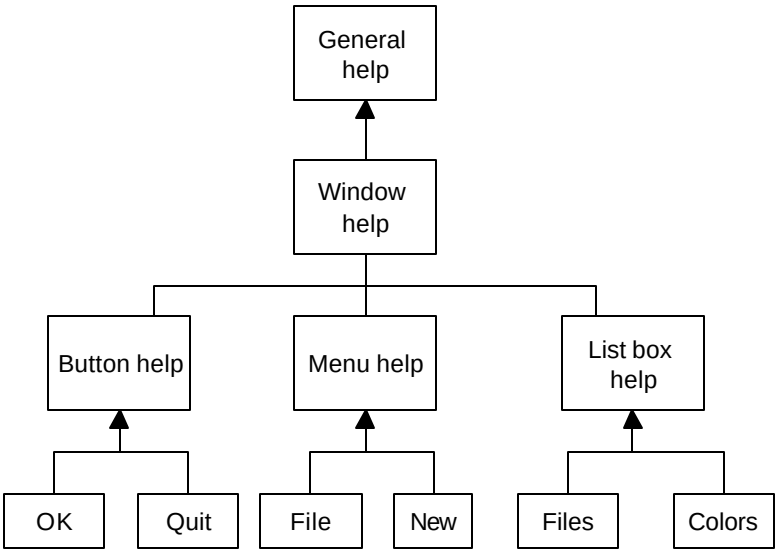


Figure 21-8– The chain of responsibility implemented as a tree structure

However, this sort of structure seems to imply that each button, or its handler, knows where to enter the chain. This can complicate the design in some cases and may preclude the need for the chain at all.

Another way of handling a tree-like structure is to have a single entry point that branches to the specific button, menu, or other widget types and then "unbranches," as previously, to more general help cases. There is little reason for that complexity—you could align the classes into a single chain, starting at the bottom, and going left to right and up a row at a time until the entire system had been traversed, as shown in Figure 21-9.

Copyright © , 2002 by James W Cooper

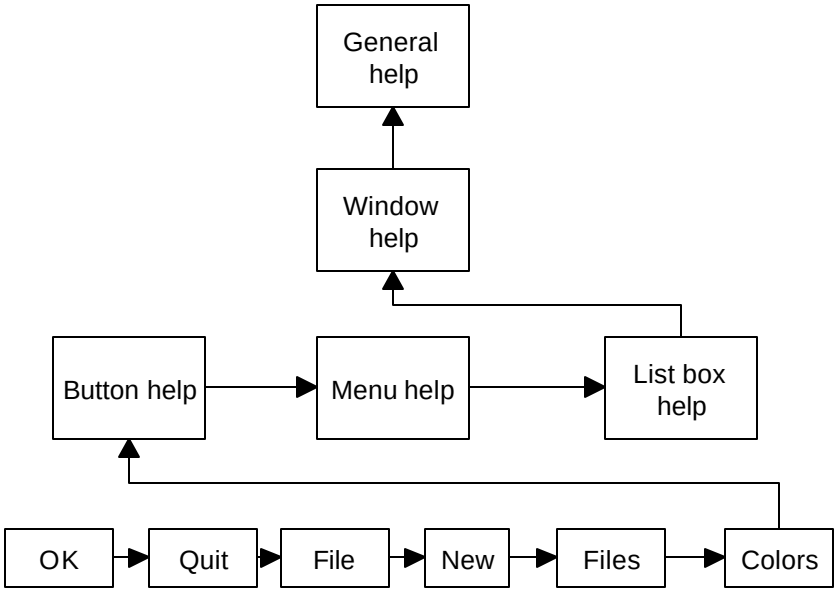


Figure 21-9 – The same chain of responsibility implemented as a linear chain

Kinds of Requests

The request or message passed along the Chain of Responsibility may well be a great deal more complicated than just the string or Control that we conveniently used on these examples. The information could include various data types or a complete object with a number of methods. Since various classes along the chain may use different properties of such a request object, you might end up designing an abstract Request type and any number of derived classes with additional methods.

Examples in C#

Under the covers, C# form windows receive various events, such as MouseMove, and then forward them to the controls the form contains. However, only the final control ever receives the message in C# whereas in some other languages, each containing control does as well. This is a

Copyright © , 2002 by James W Cooper

clear implementation of Chain of Responsibility pattern. We could also argue that, in general, the C# class inheritance structure itself exemplifies this pattern. If you call for a method to be executed in a deeply derived class, that method is passed up the inheritance chain until the first parent class containing that method is found. The fact that further parents contain other implementations of that method does not come into play.

We will also see that the Chain of Responsibility is ideal for implementing Interpreters and use one in the Interpreter pattern we discuss later.

Consequences of the Chain of Responsibility

1. The main purpose for this pattern, like a number of others, is to reduce coupling between objects. An object only needs to know how to forward the request to other objects.
2. Each C# object in the chain is self-contained. It knows nothing of the others and only need decide whether it can satisfy the request. This makes both writing each one and constructing the chain very easy.
3. You can decide whether the final object in the chain handles all requests it receives in some default fashion or just discards them. However, you do have to know which object will be last in the chain for this to be effective.
4. Finally, since C# cannot provide multiple inheritance, the basic Chain class sometimes needs to be an interface rather than an abstract class so the individual objects can inherit from another useful hierarchy, as we did here by deriving them all from Control. This disadvantage of this approach is that you often have to implement the linking, sending, and forwarding code in each module separately or, as we did here, by subclassing a concrete class that implements the Chain interface.

Thought Question

Suggest how you might use a Chain of Responsibility to implement an e-mail filter.

Copyright © , 2002 by James W Cooper

Programs on the CD-ROM

\Chain\HelpChain	program showing how a help system can be implemented
\Chain\Chain	chain of file and image displays

Copyright © , 2002 by James W Cooper

22. The Command Pattern

The Chain of Responsibility forwards requests along a chain of classes, but the Command pattern forwards a request only to a specific object. It encloses a request for a specific action inside an object and gives it a known public interface. It lets you give the client the ability to make requests without knowing anything about the actual action that will be performed and allows you to change that action without affecting the client program in any way.

Motivation

When you build a C# user interface, you provide menu items, buttons, check boxes, and so forth to allow the user to tell the program what to do. When a user selects one of these controls, the program receives a clicked event, which it receives into a special routine in the user interface. Let's suppose we build a very simple program that allows you to select the menu items File | Open, and File | Exit, and click on a button marked Red that turns the background of the window red. This program is shown in Figure 22-1.

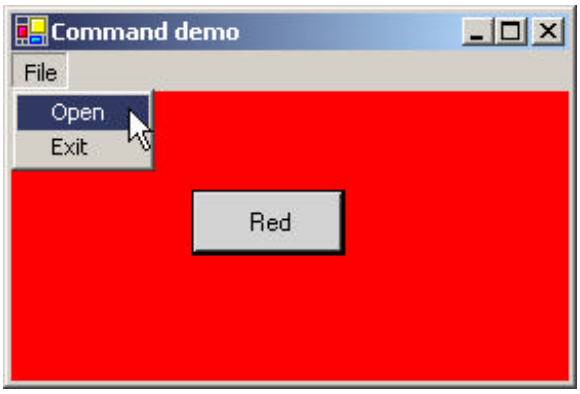


Figure 22-1 – A simple program that receives events from the button and menu items

Copyright © , 2002 by James W Cooper

The program consists of the File Menu object with the mnuOpen, and mnuExit MenuItems added to it. It also contains one button called btnRed. During the design phase, clicking on any of these items creates a little method in the Form class that gets called when the control is clicked.

As long as there are only a few menu items and buttons, this approach works fine, but when you have dozens of menu items and several buttons, the Form module code can get pretty unwieldy. In addition, we might eventually like the red command to be carried out both from the button and a menu item.

Command Objects

One way to ensure that every object receives its own commands directly is to use the Command pattern and create individual Command objects. A Command object always has an Execute() method that is called when an action occurs on that object. Most simply, a Command object implements at least the following interface.

```
public interface Command      {
      void Execute();
}
```

One objective of using this interface is to separate the user interface code from the actions the program must carry out, as shown here.

```
private void commandClick(object sender, EventArgs e) {
      Command comd = (Command)sender;
      comd.Execute ();
}
```

This event can be connected to every single user interface element that can be clicked, and each will contain its own implementation of the Execute method, by simply deriving a new clas from Button and Menuitem that supports this Command interface.

Then we can provide an Execute method for each object that carries out the desired action, thus keeping the knowledge of what to do inside the

Copyright © , 2002 by James W Cooper

object where it belongs, instead of having another part of the program make these decisions.

One important purpose of the Command pattern is to keep the program and user interface objects completely separate from the actions that they initiate. In other words, these program objects should be completely separate from each other and should not have to know how other objects work. The user interface receives a command and tells a Command object to carry out whatever duties it has been instructed to do. The UI does not and should not need to know what tasks will be executed. This decouples the UI class from the execution of specific commands, making it possible to modify or completely change the action code without changing the classes containing the user interface.

The Command object can also be used when you need to tell the program to execute the command when the resources are available rather than immediately. In such cases, you are *queuing* commands to be executed later. Finally, you can use Command objects to remember operations so you can support Undo requests.

Building Command Objects

There are several ways to go about building Command objects for a program like this, and each has some advantages. We'll start with the simplest one: creating new classes and implementing the Command interface in each. In the case of the button that turns the background red, we derive a RedButton class from Button and include an Execute method, satisfying the Command interface.

```
public class RedButton : System.Windows.Forms.Button, Command {
       //A Command button that turns the background red
       private System.ComponentModel.Container components = null;
       //-----
       public  void Execute() {
              Control c = this.Parent;
              c.BackColor =Color.Red ;
              this.BackColor =Color.LightGray  ;
       }
```

Copyright © , 2002 by James W Cooper

```
        public RedButton()          {
                InitializeComponent();
        }
```

In this implementation, we can deduce the background window by asking the button for its parent, and setting that background to red. We could just as easily have passed the Form in as an argument to the constructor.

Remember, to create a class derived from Button that you can use in the IDE environment, you create a user control, and change its inheritance from UserControl to Button and compile it. This adds an icon to the toolbox that you can drag onto the Form1 window.

To create a MenuItem that also implements the Command interface, you could use the MainMenu control on the toolbar and name it MenuBar. The designer is shown in Figure 22-2.

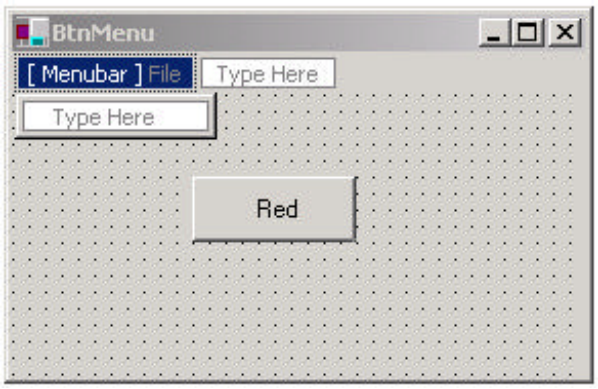


Figure 22-2– The menu designer interface

However, it is just as easy to create the MainMenu in code as we see below.

We derive the OpenMenu and ExitMenu classes from the MenuItem class. However, we have to add these in the program code, since there is no way to add them in the Form Designer.

```
private void init() {
        //create a main menu and install it
        MainMenu main = new MainMenu();
```

Copyright © , 2002 by James W Cooper

```
        this.Menu =main;

        //create a click-event handler
        EventHandler evh = new EventHandler (commandClick);
        btRed.Click += evh;        //add to existing red button

        //create a "File" top level entry
        MenuItem file = new MenuItem("File");

        //create File Open command
        FileOpen mnflo = new FileOpen ();
        mnflo.Click += evh;        //add same handler
        main.MenuItems.Add ( file );

        //create a File-Exit command
        FileExit fex = new FileExit(this);
        file.MenuItems.AddRange( new MenuItem[]{ mnflo, fex});
        fex.Click += evh;          //add same handler
}
```

Here is an example of the FileExit class.

```
public class FileExit :MenuItem, Command       {
        private Form form;
        //----------
        public FileExit(Form frm) :base ("Exit") {
                form = frm;
        }
        //----------
        public void Execute() {
                form.Close ();
        }
}
```

Then the File|Exit command will call it when you call that items Execute method. This certainly lets us simplify the user interface code, but it does require that we create and instantiate a new class for each action we want to execute.

Classes that require specific parameters to work need to have those parameters passed in the constructor or in a set method. For example, the File| Exit command requires that you pass it an instance of the Form object so it can close it.

```
//create a File-Exit command
```

Copyright © , 2002 by James W Cooper

```
FileExit fex = new FileExit(this);
```

Consequences of the Command Pattern

The main disadvantage of the Command pattern seems to be a proliferation of little classes that clutter up the program. However, even in the case where we have separate click events, we usually call little private methods to carry out the actual function. It turns out that these private methods are just about as long as our little classes, so there is frequently little difference in complexity between building the command classes and just writing more methods. The main difference is that the Command pattern produces little classes that are much more readable.

The CommandHolder Interface

Now, while it is advantageous to encapsulate the action in a Command object, binding that object into the element that causes the action (such as the menu item or button) is not exactly what the Command pattern is about. Instead, the Command object really ought to be separate from the invoking client so you can vary the invoking program and the details of the command action separately. Rather than having the command be part of the menu or button, we can make the menu and button classes *containers* for a Command object that exists separately. We thus make these UI elements implement a CommandHolder interface.

```
public interface CommandHolder    {
       Command getCommand();
       void setCommand(Command cmd);
}
```

This simple interface says that there is a way to put a command object into a class and a way to retrieve it to execute it. This is particularly important where we have several ways of calling the same action, such as when we have both a Red button and a Red menu item. In such a case, you would certainly not want the same code to be executed inside both the MenuItem

Copyright © , 2002 by James W Cooper

and the Button classes. Instead, you should fetch references to the same command object from both classes and execute that command.

Then we create CommandMenu class, which implements this interface.

```
public class CommandMenu : MenuItem, CommandHolder    {
       private Command command;
       public CommandMenu(string name):base(name)     {}
       //-----
       public void setCommand (Command comd) {
              command = comd;
       }
       //-----
        public Command getCommand () {
              return command;
       }
}
```

This actually simplifies our program. We don't have to create a separate menu class for each action we want to carry out. We just create instances of the menu and pass them different labels and Command objects.

For example, our RedCommand object takes a Form in the constructor and sets its background to red in the Execute method:

```
public class RedCommand : Command       {
       private Control window;
       //-----
       public RedCommand(Control win)          {
              window = win;
       }
       //-----
       void Command.Execute () {
              window.BackColor =Color.Red ;
       }
}
```

We can set an instance of this command into both the RedButton and the red menu item objects, as we show below.

```
private void init() {
       //create a main menu and install it
       MainMenu main = new MainMenu();
       this.Menu =main;
```

Copyright © , 2002 by James W Cooper

```
        //create a click-event handler
        //note: btRed was added in the IDE
        EventHandler evh = new EventHandler (commandClick);
        btRed.Click += evh;        //add to existing red button
        RedCommand cRed = new RedCommand (this);
        btRed.setCommand (cRed);
        //create a "File" top level entry
        MenuItem file = new CommandMenu("File");
        main.MenuItems.Add ( file );
        //create File Open command
        CommandMenu mnuFlo = new CommandMenu("Open");
        mnuFlo.setCommand (new OpenCommand ());
        mnuFlo.Click += evh;              //add same handler

        //create a Red menu item, too
        CommandMenu mnuRed = new CommandMenu("Red");
        mnuRed.setCommand(cRed);
        mnuRed.Click += evh;              //add same handler

        //create a File-Exit command
        CommandMenu mnuFex = new CommandMenu("Exit");
        mnuFex.setCommand (new ExitCommand(this));
        file.MenuItems.AddRange(
               new CommandMenu[]{ mnuFlo, mnuRed, mnuFex});
        mnuFex.Click += evh;              //add same handler
}
```

In the CommandHolder approach, we still have to create separate Command objects, but they are no longer part of the user interface classes. For example, the OpenCommand class is just this.

```
public class OpenCommand :Command       {
       public OpenCommand()
       {}
       public void Execute() {
              OpenFileDialog fd = new OpenFileDialog ();
              fd.ShowDialog ();
       }
}
```

Copyright © , 2002 by James W Cooper

Then our click event handler method needs to obtain the actual command object from the UI object that caused the action and execute that command.

```
private void commandClick(object sender, EventArgs e) {
       Command comd = ((CommandHolder)sender).getCommand ();
       comd.Execute ();
}
```

This is only slightly more complicated than our original routine and again keeps the action separate from the user interface elements. We can see this program in action in Figure 22-3:

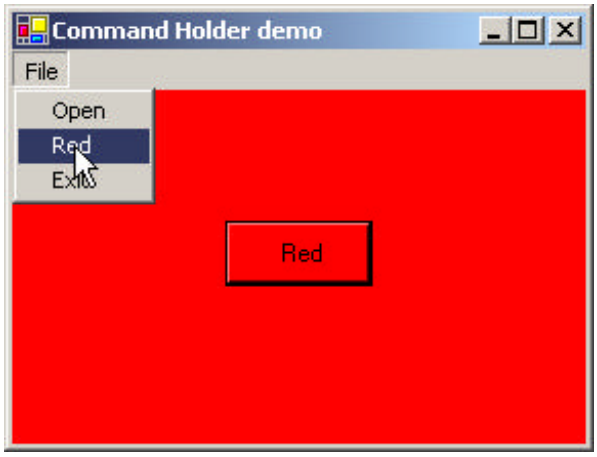


Figure 22-3 – Menu part of Command pattern using CommandHolder interface.

We can see the relations between theses classes and interfaces clearly in the UML diagram in Figure 22-4.

Copyright © , 2002 by James W Cooper